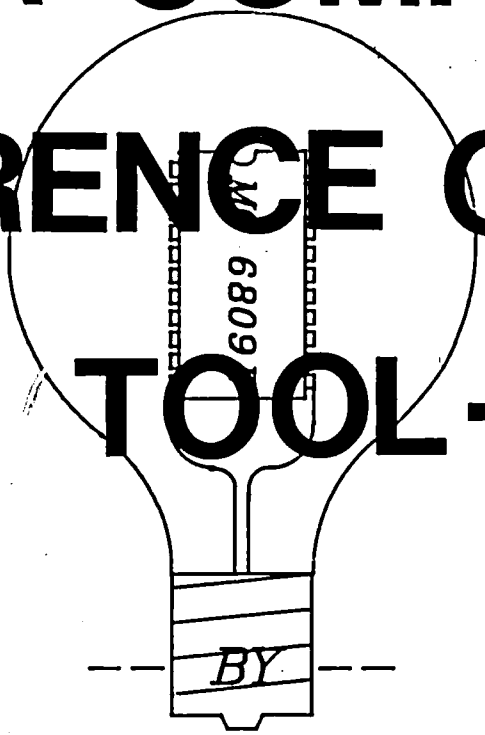
THE ULTIMATE COLOR COMPUTER REFERENCE GUIDE AND TOOL-KIT



David D. McLeod

and

Robert van der Poel

*** CMD Micro, Edmonton, Alberta, CANADA ***

THE ULTIMATE

COLOR COMPUTER

REFERENCE GUIDE AND TOOLKIT

by

David D McLeod & Robert van der Poel

--Published By --

CMD Micro Computer Services Ltd.
10447 - 124 Street
Edmonton, Alberta, Canada T5N 1R7

TABLE OF CONTENTS

INTRODUCTION

INTRODUCTION

This book is the most comprehensive BASIC reference manual currently available for the Color Computer. But that's not all! It also includes a large collection of BASIC and machine language subroutines. These routines, in addition to a series of very useful utility programs, are available separately on a pre-recorded cassette tape -- to order your copy, please use the form in the back of the text.

This book is NOT a training manual. We have not made any attempt to teach you how to write programs. Instead, we have chosen to provide a great deal of valuable information which, if used properly, will help you to write programs that run faster and better than before.

The text is divided into four sections. Section One contains the BASIC command reference. All BASIC, Extended BASIC and Disk BASIC keywords are discussed here in great detail. For easy lookup, they are all listed in alphabetical order; each one starts on its own page. The discussion for each command includes the required syntax structure, a description of the purpose of the command, any limitations which might be imposed on parameter values, and a complete listing of potential errors. For most of the commands, we have included a sub-section containing numerous hints and suggestions which will help you to make the most of command limitations.

Section Two describes numerous techniques that, if followed closely, will help you to write more concise and faster executing programs.

In Sections Three and Four, you will find a large selection of BASIC and machine language subroutines which you are free to use in your own programs. The discussion gives full details on the purpose, entry requirements and exit conditions, and sample calls for each routine. If you examine the routine listings closely, you will very likely discover some additional (undocumented) programming tricks that can easily be adapted to your own programming environment.

We believe that you will find this manual to be a valuable programming companion. Just remember that it is a reference and not a tutorial; use it as such. Happy programming!

Section One

BASIC COMMAND REFERENCE

BASIC COMMAND REFERENCE

This sections provides you with a comprehensive reference to all the statements and functions which are available in Color BASIC, Extended Color BASIC, and Disk BASIC. The keywords are contained in the text in alphabetical order for easy selection. The discussion for each keyword is broken down into 6 primary topics which are described for you below.

Syntax

Here you will find the formal syntax specification for each included keyword. We have adhered to a number of (relatively standard) rules for our syntax specifications:

1. BASIC Keywords are printed in upper case letters.
2. User-supplied information is indicated by the presence of *italics*.
3. [Optional] parts of each command are shown enclosed within square brackets.
4. Some commands allow certain parts to be repeated numerous times. In this case, any repeated part is replaced by an ellipsis ("...").

Purpose

This category provides you with a detailed description of the command--particularly with respect to how it is USUALLY used. In this description, you will find out what the command does and any peculiarities it might involve.

Arguments

Most of the BASIC keywords require specific user-supplied information in order for them to work correctly. In this area, we outline what the arguments are, and the allowable limits of the arguments.

Potential Errors

Each BASIC command can foul up in one way or another. This segment of the discussion tells you which errors are most likely to occur. We have attempted to be as exhaustive as possible with this part of the keyword description, but we cannot guarantee that we have not overlooked a potential error for a particular keyword.

We have deliberately failed to discuss syntax (?SN) and type mismatch (?TM) errors in this section (except in extraordinary cases), because they are usually quite trivial to locate and fix.

Examples

Every keyword is accompanied by a series of one or more examples of how it is NORMALLY used. We do not in any way intend that these examples should limit your imagination!

Notes/Suggestions

Except for a few extremely un-complicated keywords, you will always find one or two extra notes which (we believe) will help you to make better use of the keyword in question.

&H/&O

Syntax

&H *string expression*
&[O] *string expression*

Purpose

These two statements allow you to represent 16-bit unsigned integer numbers in either hexadecimal (&H) or octal (&O) format.

Arguments

&H - *string expression* may be from 1 to 4 characters in length; each character must be selected from the set [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]. This allows a range of hexadecimal numbers from &H0000 to &HFFFF (0 to 65535).

&[O] - *string expression* may be from 1 to 6 characters in length; each character must be selected from the set [0, 1, 2, 3, 4, 5, 6, 7, 8]. This allows a range of octal numbers from &O000000 to &O177777 (0 to 65535). Note that element "8" should not be included in the set of valid octal digits; however, a bug in BASIC allows it to exist. If the "O" is deleted, then the argument is evaluated as octal notation.

Potential Errors

OV - the value of *string expression* exceeds 65535.

Examples

```
POKE &H00B2, &H33

A = PEEK (&H0095)

A = &O10

A = &10
```

Notes/Suggestions

Hexadecimal notation is much more common than octal notation today because of modernized computer architecture. Since most processors have word sizes that are a multiple of 4 bits instead of 3 bits, octal notation is virtually obsolete.

There are a couple of reasons why hexadecimal notation is

to be preferred over decimal notation (but not in all applications, of course). First of all, BASIC is able to process numbers in this format about 50% faster than it can process the same numbers in decimal. This is because the conversion from ASCII-coded hexadecimal to floating-point binary is a very simple process, while the same conversion of ASCII-coded decimal is quite a bit more time consuming. Second, machine-language code in DATA statements is much easier to disassemble when it is kept in hexadecimal format. For these reasons, any time your application does not require floating-point accuracy and the magnitude of the data is between 0 and 65535, you should use hexadecimal notation. Bear in mind, however, that this notation will always require 2 extra bytes of memory whenever it is used.

The two statements A = &H1000 and A = VAL ("&H" + F$) (where F$ = "1000") perform exactly the same function. In both cases, A will have the value 4096. But note that the first statement is faster than the second. The only time the second form should be used is when you want to convert a string that was formulated by the HEX$ command.

ABS

Syntax

numeric variable = ABS *(expression)*

Purpose

This function returns the absolute value, or magnitude, of *expression*. This means that the sign value of *expression* is ignored and the result is always a positive number.

Arguments

expression must evaluate to any positive or negative number in the range 10^{-38} to 10^{+38}.

Potential Errors

OV - the magnitude of *expression* exceeds 10^{+38}.

Examples

```
A = ABS (2 * X + SIN (3.14159))
```

Notes/Suggestions

Use ABS and SGN together to separate any number into its sign and magnitude parts.

AND

SEE *LOGICAL OPERATORS*

ARITHMETIC OPERATORS

Syntax

1. Unary Plus --
numeric variable = + *expression*

2. Unary Minus --
numeric variable = - *expression*

3. Exponentiation --
numeric variable = *expression* ↑ *expression*

4. Multiplication --
numeric variable = *expression* * *expression*

5. Division --
numeric variable = *expression* / *expression*

6. Addition --
numeric variable = *expression* + *expression*

7. Subtraction --
numeric variable = *expression* - *expression*

8. String Concatenation --
string variable = *string1* + *string2*

Purpose

These operators are used in performing arithmetic calculations. Items 1 to 7 in the syntax list are ordered according to the precedence value of the operation; thus, unless parentheses are used to force a different order of precedence, unary plus and minus will be carried out first, followed by exponentiation, followed by multiplication and division, followed finally by addition and subtraction.

Arguments

expression may evaluate, for the most part, to any positive or negative value in the range 10^{-38} to 10^{+38}, depending on the operations actually performed, as long as the result that is passed to *numeric variable* is in the same range.

string1 and *string2* may be any string of from 0 to 255 characters, each of which may have any ASCII value between 0 and 255, as long as the result which is passed to *string variable* is not longer than 255 characters.

Potential Errors

/0 - a "division by zero" has been attempted; the *expression* immediately following a "/" (division) sign is equal to zero.

LS - you have attempted a string concatenation which would normally produce a string longer than 255 characters.

OS - a string resulting from a concatenation is longer than the amount of reserved string space will allow.

OV - the result of an operation is outside the allowable range of 10^{-38} to 10^{+38}.

ST - a formula (a series of concatenations) for producing a string is too complex for BASIC to handle.

Examples

```
A = 2 / 3 * B

A = 2 / (3 * B)

A$ = A$ + STRING$ (10,"*") + A$
```

Notes/Suggestions

You must always be aware of the order in which operations will be performed, especially when your expression consists of a series of different operations. Consider the first two examples above, each of which produces a different result. In the first example, "2 / 3" is evaluated first, and then the result is multiplied by B. In the second example, however, the parentheses force the computer to perform the "3 * B" operation first and to use the result as the denominator for the division into 2.

The use of parentheses in numeric operations makes it quite easy to alter the order of precedence for any series of operations. But remember that each parenthesis takes up an additional byte of memory and actually slows down the speed of the operation, so use parentheses only when they are absolutely necessary.

AS

SEE *FIELD*

ASC

Syntax

numeric variable = ASC *(string expression)*

Purpose

This function returns the ASCII code of the first encountered character in *string expression*.

Arguments

string expression may be a string variable name, a string literal, or a formula which evaluates to a valid character string.

Potential Errors

FC - *string expression* is a null (zero length) string.

Examples

```
A = ASC (T$)

A = ASC ("HI")

IF ASC (Q$) = 8 THEN ...
```

Notes/Suggestions

The third example above could have been written as "IF Q$ = CHR$ (8) THEN ..." and would produce the same result. Note, however, that for an "IF-THEN" application, the *ASC* function is marginally faster than the *CHR$* function.

ATN

Syntax

numeric variable = ATN (*expression*)

Purpose

This function returns the arctangent of *expression* in radians. (Arctangent is defined as the angle whose tangent is *expression*.)

Arguments

expression may be any signed value whose magnitude is in the range 10^{-38} to 10^{+38}.

Potential Errors

OV - the magnitude of *expression*.exceeds 10^{+38}

Examples

```
A = ATN (X + Y)

IF ATN (A) < PI/3 THEN ...
```

AUDIO

Syntax

AUDIO *argument*

Purpose

This statement allows you to enable or disable sound from the cassette player to the T.V. speaker. It can be used in conjunction with the MOTOR statement to allow the simultaneous playback of music or other recorded data while a BASIC program is executing. AUDIO is automatically turned off whenever BASIC encounters an error.

Arguments

The only valid arguments are *ON* or *OFF*.

Potential Errors

None.

Examples

```
AUDIO ON

AUDIO OFF
```

BACKUP

Syntax

BACKUP *source [TO destination]*

Purpose

This statement allows you to duplicate the contents of the diskette in the *source* drive. If no *destination* drive is specified or *destination* is the same as *source*, you are prompted as necessary to swap diskettes in the *source* drive. A NEW is automatically performed when you execute a BACKUP; i.e., any BASIC program will be removed from memory.

Arguments

Both *source* and *destination* must be expressions that evaluate to an integer value from 0 to 3.

Potential Errors

DN - drive number either unspecified or out of range.

IO - *source* disk is unreadable or *destination* disk is flawed or unformatted.

VF - VERIFY is turned on; the last written sector cannot be read back. (This is a specialized IO error.)

Examples

```
BACKUP 0 TO 3

BACKUP 1
```

CHR$

Syntax

string variable = CHR$ *(expression)*

Purpose

This function creates a string consisting of one character whose ASCII code value is specified by *expression*.

Arguments

Since the character created must have a valid ASCII code, *expression* must evaluate to a number in the range 0 to 255.

Potential Errors

FC - *expression* is outside the range 0 to 255.

Examples

```
A$ = CHR$ (&H3F)

A$ = CHR$ (128) + "ABC" + CHR$ (129)

IF A$ = CHR$ (Q) THEN ...
```

Notes/Suggestions

In the third example above, the same outcome can be achieved by "IF ASC (A$) = Q THEN ...", which executes slightly faster.

CIRCLE

Syntax

CIRCLE *(X, Y), R [, C, HW, start, end]*

Purpose

This statement causes a circle of radius *R* to be drawn at centre point *(X, Y)*. If specified, *C* defines the color of the points in the circle (BASIC's default is the current foreground color); *HW* defines the height-width ratio, which allows for horizontal or vertical elliptic shapes (default value is 1); *start* defines the start point of a circular arc (default value is 0); *end* defines the end point of a circular arc (default value is 1).

Arguments

X - a valid horizontal co-ordinate from 0 to 255

Y - a valid vertical co-ordinate from 0 to 191

R - may be any positive value such that R * HW < 65536

C - may be any value from 0 to 8

HW - may be any value from 0 to 255 (values between 0 and 1 produce horizontal ellipses; values above 1 produce vertical ellipses)

start - may be any fractional value from 0 to 1

end - may be any fractional value from 0 to 1 (*start* and *end* represent fractions of 360 degrees or 2 * PI radians)

Potential Errors

FC - one or more of the above arguments is (are) beyond allowable limits.

Examples

```
CIRCLE (X, Y), 100, , .95, , .9

CIRCLE (127, 95), R
```

Notes/Suggestions

It is possible to cause unusual patterns to be created by POKEing color mask values directly into memory location &H00B2. However, if you do this, do not include a color argument in the CIRCLE statement. (This same technique works for DRAW, LINE, and PAINT as well.)

CLEAR

Syntax

CLEAR *[string space] [, highest address]*

Purpose

This statement performs two separate functions. First, it causes all BASIC variables to be reset--that is, numeric variables are all set to zero, and string variables are all set to null. Second, it allows you to reserve memory space for string allocation and/or for other protected usage (usually machine-language programs or sub-routines). Note that the syntax for this command is a little unusual--if *highest address* is present, *string space* must also be present. The converse is not true.

Arguments

string space must be a numeric expression that defines how many memory locations are to be reserved for string allocation. BASIC's default value is 200 bytes.

highest address must be a numeric expression that defines the last memory location BASIC is permitted to use. BASIC's default value will be the highest available RAM location (&H3FFF for a 16K machine, &H7FFF for a 32K machine).

Potential Errors

FC - *string space* is bigger than the physical RAM limit, or *highest address* is not between &H0000 and &HFFFF (0 to 65535).

OM - one of the arguments has caused available program memory to drop below 60 bytes or *highest address* is above the physical RAM limit.

Examples

```
CLEAR

CLEAR 50

CLEAR 200, &H3F00
```

Notes/Suggestions

It is possible to cause an irreversible OM error with this command if you are not careful. Try the following experiment for yourself immediately after power-up (but not when you have a program in memory!) Type CLEAR 0 <ENTER>, followed by CLEAR MEM-58 <ENTER>. These two commands (both of which should terminate with the normal OK prompt) will cause all but 60 bytes of free memory to be reserved as string space. You can verify this by typing PRINT MEM <ENTER>. Note, however, that any subsequent command that involves any kind of processing will result in an OM error. The only way to recover from this error is to power down and power up again (RESET will not work). This example should demonstrate the caution required when using the CLEAR command.

CLOAD

Syntax

CLOAD *[filename]*

Purpose

This statement allows you to load a BASIC program from cassette tape. If *filename* is not provided, the first encountered program will be loaded. Once loaded, the program may be executed by means of the RUN command.

Arguments

filename may be any character string of from 0 to 8 characters in length. Each character may have any ASCII value from 0 to 255. Note that a null string is treated as if no *filename* were specified and that a filename that exceeds 8 characters in length will be truncated to the 8 character limit.

Potential Errors

FM - the encountered file is not a BASIC program.

IO - the program is loading into bad memory (possibly ROM) or the tape is unreadable (possibly because the volume is set incorrectly).

Examples

```
CLOAD

CLOAD "PROGRAM"

CLOAD F$
```

Notes/Suggestions

When the CLOAD statement appears in a program line after an ELSE or THEN statement, CLOAD must be preceded by a colon (:); otherwise, an SN ERROR will occur.

CLOADM

Syntax

CLOADM *[filename] [, offset]*

Purpose

This statement allows you to load a machine-language program into memory. If *filename* is not specified, the next encountered program will be loaded. If *offset* is supplied, this value will be added to the start, end and entry addresses of the program, thus causing the program to load into a different place in memory. Once loaded, the program may be executed by means of the EXEC command.

Arguments

filename may be any character string of from 0 to 8 characters in length. Each character may have any ASCII value from 0 to 255. Note that a null string is treated as if no *filename* were specified and that a filename that exceeds 8 characters in length will be truncated to the 8 character limit.

offset may be any numeric expression which evaluates to between &H0000 and &HFFFF (0 and 65535). Note that *offset* may only be present if *filename* is specified or a null string ("") is used.

Potential Errors

FC - the specified *offset* is out of range.

FM - the encountered file is not a machine-language program.

IO - the program is loading into bad memory (possibly ROM) or the tape is unreadable (possibly because the volume is set incorrectly).

Examples

```
CLOADM

CLOADM "PROGRAM", &H1234

CLOADM F$

CLOADM "", &H1234
```

Notes/Suggestions

CLOADM may be used within a BASIC program to load a co-resident machine-language program. However, you should use the CLEAR statement to protect memory before loading.

When the CLOADM statement appears in a program line after an ELSE or THEN statement, CLOADM must be preceded by a colon (:); otherwise, an SN ERROR will occur.

CLOSE

Syntax

CLOSE *[[#] buffer] [,[#] buffer] [, ...]*

Purpose

This statement causes open files to be closed in the order specified. If no *buffer #* is supplied, all files will be closed in the reverse sequence in which they were opened. Note that CLOSEing a file that was opened for <O>utput may result in a final block of data being written to the specified *buffer #*.

Arguments

buffer # may be an expression which evaluates to a number between -2 and 15, which correspond to the following devices:

```
-2    - printer (RS-232)
-1    - cassette
 0    - screen/keyboard
 1-15 - disk
```

Potential Errors

DF - the CLOSE statement resulted in a write attempt to disk, at which time the disk was found to be full.

FC - *buffer #* is out of range.

Examples

```
CLOSE

CLOSE 1

CLOSE #2, #-1, #3
```

Notes/Suggestions

When closing multiple open files (especially when one or more of them is open for <O>utput), always close them in the reverse order to which they were opened. If you do not do this, final data blocks may be written to the wrong buffer, causing the file's integrity to be destroyed. This is particularly dangerous when your files happen to be on different disk drives, in which case both directory allocation tables may also be incorrectly written!

CLS

Syntax

CLS *[color code]*

Purpose

This statement causes the normal text screen to be cleared to the specified color, and updates the print position so that the next PRINT will occur in the upper left corner of the screen. When no *color code* is specified, the screen is actually filled with modified spaces (value &H60 or 96). When *color code* is specified, the screen is filled with a value that represents a corresponding graphic character.

Arguments

color code may be an expression which has any value from 0 to 255, although only values from 0 to 8 will actually cause a color to be selected. Values from 9 to 255 will cause the word "MICROSOFT" to appear in the upper left corner of the screen, but this is not a bona-fide error, since your program still retains control. *color code* values from 0 to 8 result in the following colors:

0 - (value &H80 or 128) - black
1 - (value &H8F or 143) - green
2 - (value &H9F or 159) - yellow
3 - (value &HAF or 175) - blue
4 - (value &HBF or 191) - red
5 - (value &HCF or 207) - buff
6 - (value &HDF or 223) - cyan
7 - (value &HEF or 239) - magenta
8 - (value &HFF or 255) - orange

Potential Errors

FC - the specified *color code* is out of range.

Examples

```
CLS

CLS 3

CLS (X)
```

Notes/Suggestions

Although it is permitted to enclose the *color code* in parentheses, this practice is not required. Consider the fact that each parentesis requires an additional byte of memory space within your program...

COLOR

Syntax

COLOR *[foreground] [, background]*

Purpose

This statement allows you to specify the *foreground* color (used by CIRCLE, DRAW, LINE, PAINT, and PSET) and/or the *background* color (used by PCLS and PRESET). Only the argument(s) specified will be changed.

Arguments

Both *foreground* and *background* may be expressions which evaluate to numbers between 0 and 8.

Potential Errors

FC - one or both of the specified color codes is (are) out of range.

Examples

```
COLOR 1, 3

COLOR 2

COLOR , X
```

Notes/Suggestions

It is possible to create some very unusual and striking patterns by POKEing values directly into memory before using any of Extended BASIC's graphic commands. To do so, POKE any value from 0 to 255 into location &H00B2 to affect *foreground* color, or into location &H00B3 to affect *background* color. This technique precludes the need for the COLOR command.

CONT

Syntax

CONT

Purpose

This statement is used in debugging BASIC programs. It allows you to continue program execution after you have pressed the <BREAK> key, or the program has encountered a STOP command. It will work as long as no BASIC error was produced, and you have not EDITed any BASIC lines.

Arguments

None.

Potential Errors

CN - program execution cannot be resumed because either a BASIC error occurred or you EDITed a BASIC line.

Examples

CONT

COPY

Syntax

COPY *source [*TO *destination]*

Purpose

This statement allows you to COPY a disk file specification from one place to another. If no *destination* is supplied, you are automatically prompted to change diskettes as required.

Arguments

source and *destination* must be valid disk file specifications. A filespec consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (".), slash ("/"), or colon (":").

extension - any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/").

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

Potential Errors

AE - *destination* filespec already exists.

DF - there is not enough room on the *destination* disk for the new file.

FN - invalid filename or extension.

IO - *source* diskette is unreadable or *destination* diskette is flawed or unformatted.

VF - VERIFY is turned on; the last written sector cannot be read back. (This is a specialized IO error.)

WP - *destination* diskette has a write-protect tab on it.

Examples

```
COPY "PROGRAM/BAS"

COPY "FILE.EXT:1" TO "0:NEWFILE.EXT"

COPY F1$ TO F2$
```

Notes/Suggestions

If you have Disk BASIC Version 1.0 in your system, you may never get an "?IO ERROR" when executing this command. Due to a bug in the ROM, it is likely that your computer will hang up instead. If this happens, you can regain control by pressing RESET, or by turning the computer off and then on again. This problem does not seem to occur in computers with Disk BASIC Version 1.1.

The COPY command may be used as an alternative to the BACKUP command (See our utility "DSKMAN" for examples of this). Although quite a bit more time consuming, COPY will allow you to completely clean up an old disk where several files have been KILLed and the disk over-written. Additionally, COPY can be used within a BASIC program, whereas BACKUP cannot.

An existing file can be duplicated on the same disk -- all you need to do is use different filenames for *source* and *destination* with the same drive numbers.

COS

Syntax

numeric variable = COS *(expression)*

Purpose

This function returns the cosine of *expression*, which is assumed to be a radian angle. The returned value is always a rational number between -1 and +1.

Arguments

expression may be any positive or negative value in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
A = COS (2 * PI / 7)

A = COS ((90 * (2 * PI))/ 360)

IF COS (X) < 0 THEN ...
```

Notes/Suggestions

Given that PI is defined as 3.1415928, given that in any circle 360 degrees = 2 * PI radians, and given an angle A in degrees, it is possible to convert to a radian angle R by means of the formula

```
R = (A * 2 * PI) / 360
  = 0.01745329 * A.
```

Similarly, given any radian angle R, A can be obtained from the formula

```
A = (360 * R) / (2 * PI)
  = 57.29578 * R.
```

CSAVE

Syntax

CSAVE *filename*

Purpose

This statement causes the BASIC program currently in memory to be written out to cassette tape. If this command is included within a program and it follows a THEN or ELSE statement, it MUST be preceded by a colon.

Arguments

filename may be any character string of from 0 to 8 characters in length. Each character may have any ASCII value from 0 to 255. If the filename exceeds 8 characters in length, it will be truncated to the 8 character limit.

Potential Errors

None.

Examples

```
CSAVE "PROGRAM"

CSAVE F$
```

CSAVEM

Syntax

CSAVEM *filename, start, end, entry*

Purpose

This statement causes a machine-language file to be written out to cassette tape. It duplicates the contents of memory from *start* to *end* inclusive. *entry* must be supplied so that when the file is re-loaded, BASIC will be able to EXECute the program. If this statement is used within a BASIC program and it follows a THEN or ELSE statement, it MUST be preceded by a colon.

Arguments

filename may be any character string of from 0 to 8 characters in length. Each character may have any ASCII value from 0 to 255. If the filename exceeds 8 characters in length, it will be truncated to the 8 character limit.

start, *end*, and *entry* must be valid addresses in the range &H0000 to &HFFFF (0 to 65535).

Potential Errors

FC - *start*, *end*, or *entry* values are out of range.

Examples

```
CSAVEM "PROGRAM", 1234, 2345, 1234

CSAVEM "FILE", &H1000, &H1FFF, &H1100

CSAVEM F$, A, B, C
```

CVN

Syntax

numeric variable = CVN *(string expression)*

Purpose

This statement converts a 5-character string, which represents a floating-point number, so that it can be displayed. Although it is used primarily to decode data that was encoded by the MKN$ command, it will work on any 5-character string.

Arguments

string expression must be exactly 5 characters in length, but each character may have any ASCII value from 0 to 255.

Potential Errors

FC - *string expression* is not 5 bytes long.

Examples

```
A = CVN (A$)

A = CVN ("HELLO")

IF CVN (X$ + Y$) < 1E-10 THEN ...
```

DATA

Syntax

DATA *[data item] [, data item] [, ...]*

Purpose

This statement permits the construction of one or more lists of constant data that are accessed by means of the READ statement. Depending on the nature of each *data item*, data may be read into either a numeric variable or a string variable.

Arguments

data items must be separated by commas. In the case of numeric data, *data item* must be written as a number in a format that is acceptable to BASIC (e.g. 4096, 4.096E+3, or &H1000.) In the case of string data, *data item* may be any combination of characters, except that colons and commas may not be part of the string unless the entire string is surrounded by quotation marks.

Potential Errors

None.

Examples

```
DATA 1, 2, 3

DATA "1, 2, 3"
```

Notes/Suggestions

The DATA statement is a very peculiar instruction, in that it is possible to append additional statements to the end of a data list and to both execute the added instructions and read them as if they were actual data entries! Consider the following examples:

```
00010 DATA
      :CLS
      :POKE &H500, &HFF
      :A = PEEK (&H500)
00020 FOR I = 1 TO 5
      :READ B
      :PRINT B
      :NEXT I
```

```
00010 DATA
      :CLS
      :POKE &H500, &HFF
      :A = PEEK (&H500)
00020 FOR I = 1 TO 5
      :READ B$
      :PRINT B$
      :NEXT I
```

These two short BASIC programs are identical except that example #2 reads the data into string variable B$ instead of numeric variable B. Curiously, example #2 will RUN without error, but example #1 produces a "?SN ERROR IN 10". Note, however, that this error does not occur until after line 10 has been completely executed. This is verified by the fact that the screen clears, an orange graphic block appears on the centre line, and variable A has been assigned the value 255. What this strange error means, then, is that no error occurred until the READ statement was encountered in line 20. But what about example #2? If you RUN it you will notice that B$ takes on the value of the tokenized BASIC statements one at a time for each pass through the READ loop. Essentially, this means that both colons and commas are recognized as delimiters. If you remove the colon between DATA and CLS in example #2, the screen will not clear, but the remainder of the instructions will be executed, and you will get an "?OD ERROR IN 20".

We have not discovered any particular usefulness for this odd quirk in the DATA statement, but you may be able to figure out some purpose for it.

DEF

Syntax

DEF FN *name (numeric variable) = expression*
DEF USR *[number] = address*

Purpose

This statement allows you to define one or more mathematical functions and up to 10 machine-language sub-routine functions.

Arguments

a. DEF USR

number may be any decimal digit from 0 to 9. If not specified, zero is automatically assumed.

address may be an expression which evaluates to a 16-bit integer number between &H0000 and &HFFFF (0 and 65535).

b. DEF FN

name may be any character string consisting of the upper case letters of the alphabet and/or the digits from 0 to 9. The first character in the name must be an upper-case letter. If the string consists of more than 2 characters, only the first 2 characters will be recognized by BASIC. Note that these are the same rules that apply to all *numeric variables*.

numeric variable must be specified even if the function does not require a parameter, in which case it becomes a "dummy parameter".

expression may be any numeric or string expression as long as it evaluates to a positive or negative number between 10^{-38} and 10^{+38}.

Potential Errors

ID - DEF can only be used within a BASIC program; it cannot be used as a direct statement.

OV - *address* is out of range.

Examples

```
DEF USR = &H1000

DEF USR0 = 4096

DEF FN A (X) = 2 * X + COS (X)

DEF FN B (Y) = LOG (Y) + LOG (2 * FN A (X))

DEF FN C (A) = INSTR (A, A$, ".")
```

DEL

Syntax

DEL *[startline] [-] [endline]*

Purpose

This statement causes BASIC program lines to be DELeted from the body of the program.

Arguments

Both *startline* and *endline* must be legitimate line numbers (i.e. from 0 to 63999), even though the referenced lines do not have to actually exist in memory. Note that one or both of the arguments must be present--that is, either *startline* or *endline* (or both, separated by a hyphen) must be specified.

Potential Errors

FC - no line number has been supplied.

Examples

```
DEL 10

DEL 10 - 50

DEL - 50

DEL -          -- same effect as a NEW
```

DIM

Syntax

DIM variable [(size [, ...])] [, variable] [, ...]

Purpose

This statement allocates sufficient memory space for each *variable* in the list, and initializes the value to zero for a numeric variable or null for a string variable. The variable list may refer to all variable types including simple variables and array variables.

Arguments

variable may be any legitimate variable name consisting of the upper-case letters of the alphabet and the digits from 0 to 9. The first character must always be an upper-case letter; if the name exceeds 2 characters in length, only the first 2 will be recognized by BASIC. If the variable name is terminated by a "$", the variable will be recognized as a string variable.

size is used to specify the number of elements in each dimension of an array variable. The limits on the value of *size* will depend on the amount of free memory available, and on the number of dimensions in the array. When the program is run, the actual number of elements will be *size + 1*.

Potential Errors

BS - the value of *size* exceeds 13106 (32K machine).

DD - the specified *variable* has been dimensioned before (variables cannot be dimensioned twice in the same program unless a CLEAR statement has been executed first).

FC - the value of *size* exceeds 32767.

OM - the value of *size* is too large for the amount of free memory remaining.

Examples

```
DIM A, B, A$

DIM A (6, 6, 3)
```

Notes/Suggestions

Whenever a new variable is introduced in a BASIC program, the interpreter automatically performs a DIM, which means that a search is done to see if the variable already exists. If not, then space must be allocated for the new variable. This is particularly noteworthy when the new variable is a simple variable, in which case all previously defined array variables must be moved upward in memory to make room for the new definition. This, of course, takes time.

In order to eliminate or reduce this time, it is good practice to DIMension all variables just once in an initialization segment of your program. Since BASIC must search for variable names as they are referenced, it also makes good sense to place the most frequently used variables at the beginning of the search area. This can be accomplished by DIMensioning them in the desired order. Determining the frequency of variable usage is a difficult and time-consuming task, and for this reason, we have developed a utility program ("VARLIST", available separately on tape) which will do the job for you. Once you have the ordered list of variable names, you can construct a DIM statement which will preset the entire variable space for you. This DIM statement should list the simple variables first, followed by array variables.

DIR

Syntax

DIR *[drive]*

Purpose

This statement allows you to read the contents of the DIRectory sectors on the diskette in the specified drive.

Arguments

drive must evaluate to a number between 0 and 3. If not specified, the current default drive will be used.

Potential Errors

DN - *drive* is out of range.

FS - one of the entries in the directory has been improperly written to disk and therefore, BASIC is unable to determine the nature of the file. This is most likely to happen after a file has been opened for output, but is never properly closed.

IO - the disk is not properly inserted in the drive, the door is not closed, or the data on the disk is unreadable.

Examples

```
DIR

DIR 2
```

Notes/Suggestions

BASIC makes no explicit allowance for sending the output of the DIR command to the printer. This can be accomplished, however, by the following statements, which replace the above examples:

```
POKE &H006F, &HFE: DIR <ENTER>

POKE &H006F, &HFE: DIR 2 <ENTER>
```

DLOAD/DLOADM

Syntax

DLOAD *[filename] [, baud]*
DLOADM *[filename] [, baud] [, offset]*

Purpose

These statements allow you download a BASIC or machine-language program from another computer which acts as a host machine. In order for the commands to work, your computer must be hooked up to the host via the RS-232 ports (this may be a direct connection through a network controller or it may be through a pair of telephone modems), and the host must be running a controller program which will recognize and respond to special serial transmission codes from your computer.

Arguments

filename may be any character string of from 0 to 8 characters in length. Each character may have any ASCII value from 0 to 255. Note that a null string is treated as if no *filename* were specified, in which case the host computer decides which file will be transmitted, and that a filename that exceeds 8 characters in length will be truncated to the 8 character limit.

baud must be either 0, which corresponds to a baud rate of 300 bits per second, or 1, which corresponds to 1200 bits per second. If no *baud* value is specified, the last defined baud rate will be used. On power-up, BASIC sets the baud to 300.

offset must evaluate to an integer number from &H0000 to &HFFFF (0 to 65535).

Potential Errors

FM - you have requested a BASIC program file by means of the *DLOADM* command, or you have requested a machine-language file by means of the *DLOAD* command.

IO - your computer is not able to receive valid data from the host, even after 5 separate attempts.

NE - the requested file is not present in the host's memory.

Examples

```
DLOAD "PROGRAM", 0

DLOADM , , &H1000
```

Notes/Suggestions

If you have only Extended BASIC in your computer, you will discover that the DLOADM command contains a small bug, in that you cannot specify a literal filename. For example, if you type DLOADM "MLPROG", the computer will respond with a "?TM ERROR". This bug can be overcome by replacing the command with the following statements:

```
F$ = "MLPROG": DLOADM F$
```

If you have Disk BASIC in your computer, this bug will not occur.

DOS

Syntax

DOS

Purpose

This statement is made available in Disk BASIC Version 1.1. Its purpose is to permit the loading and auto-execution of a machine-language program (usually an operating system, such as OS-9). It causes the computer to read in all of track #34 from the disk in drive #0 and to store the data in memory starting at &H2600. If the first two bytes of this data are equal to the ASCII values for "OS", then control is passed to the program starting at &H2602; otherwise, a RESET is performed.

Arguments

None.

Potential Errors

IO - one or more sectors on Track #34 is (are) unreadable.

Examples

DOS

Notes/Suggestions

Beware of using this command when you have an unsaved BASIC program in memory. Obviously, if there is a legitimate program on Track #34, you will lose control of the computer. But, even if the data on Track #34 is not valid, your program will likely be destroyed, especially if the computer is configured at PCLEAR 4 or higher.

Whether you have a program in memory or not, following a failed DOS command, you will have to perform a NEW in order to fix BASIC's pointers. DOS does not do this for you.

DRAW

Syntax

DRAW *string*

Purpose

This statement is used to draw lines on the high resolution graphics screens. By varying the angle, scale, color and direction of the lines, you can create quite complex figures.

Arguments

In the following discussion, the word "cursor" is used to indicate the position at which the next DRAW will occur. No cursor is used by the statement, but the word is used here as a convenient descriptor.

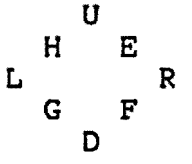
String can include any of the following:

Move cursor - "M" followed by the X,Y co-ordinates. The two co-ordinates must be separated from each other by a comma. Each co-ordinate must be an integer between 0 and 255. If a number greater than 191 is specified for the Y co-ordinate, the line will be drawn at the edge of the screen.

Blank move - "BM" followed by the X,Y co-ordinates (see above).

Move offset - "BM" or "M" followed by the X,Y co-ordinates, with each co-ordinate preceded by a "+" or "-". This will move the cursor the number of points specified by the arguments from the current cursor position.

Draw line - the letters U, D, L, R, E, F, G, H will draw a line in the direction indicated in the diagram below. The number of points drawn is specified by the number following the letter. If no number is included, a default of 1 is used. All the parameters must be between 0 and 255.



Draw blank line - "B" preceding a letter in the above diagram will move the cursor in the the direction indicated, without a line being drawn.

No update - "N" preceding a letter in the above diagram will cause the line to be drawn without the cursor position being updated.

Color - "C" followed by a number between 0 and 8 will change the foreground color to the one specified by the number.

Angle - "A" followed by an integer between 0 and 4 will cause the angle at which subsequent lines are drawn to be changed. On powerup the default value of 0 is used. Any additional draw commands will be executed at the new angle. The following angle arguments are valid:

0 - 0 degrees
1 - 90 degrees
2 - 180 degrees
3 - 270 degrees

If an "A2" is included, a "U10" would draw a line down 10 points, rather than in the up direction expected.

Scale - "S" followed by an integer between 1 and 62 will change the scale at which subsequent lines are drawn. On powerup the default value of 4 is used. The argument represents a fraction with 4 as its denominator. For example, a "S1" will draw the subsequent commands in 1/4 scale and a "S12" will draw the subsequent commands in 12/4 (or 3X) scale.

Delimiters - any of the arguments can be separated from each other by spaces or semicolons ";". Delimiters are optional in the above operations and should generally not be used due to the memory they consume.

To pass a numeric parameter to *string*, the following syntax is allowed:

operation = variable;

where *operation* is the letter M, U, D, L, R, E, F, G, H, S, C, or A and *variable* is any numeric variable. Note that a semicolon must follow *variable*, even if it is at the end of a string.

Substrings can be executed within a DRAW statement with the following:

X*substring;*

where *substring* is a previously defined string variable. Note that a semicolon must follow the variable name, even if it is at the end of a string.

Potential Errors

FC - any error in the syntax of the string will result in the FC error.

Examples

```
DRAW "BM120,120;U30;L30;D30;R30"

DRAW "BM120,120ULNL20BD10G44"

DRAW A$

DRAW "BM=X;,=Y;,U=A;L33"

DRAW "BM30,30M+10,-10S12C3U3D3A1U3D3"
```

Notes/Suggestions

The "=" syntax is not described in the Radio Shack documentation for Extended Color Basic and deserves a bit more explanation. If you wish to pass a variable to a DRAW command, Going Ahead with Extended Color Basic suggests that you convert the variable to a string as in the following example:

```
10 FOR V = 0 TO 3
20 A$ = "A" + STR$ (V)
30 DRAW A$ + "U4R4D4L4"
40 NEXT V
```

A much simpler, and faster, loop can be constucted using the "=" syntax, as the following example illustrates:

```
10 FOR V = 0 TO 3
20 DRAW "A=V;U4R4D4L4"
30 NEXT V
```

Note that the variable name can be any 2 letter combination -- V is used here for angle. When the "=" is encountered in the string, the angle takes on the value of "=V;". The first time the above loop is interpreted it would be equivalent to "A0U4R4D4L4"; the last time through the loop, "A3U4R4D4L4".

With the same technique, you can pass co-ordinate values as well. In the following example, the figure will be drawn in five different locations on the screen:

```
10 A$ = "U4R4D4L4"
20 FOR T = 1 TO 5
30 READ X, Y
40 DRAW "BM=X;,=Y;" + A$
50 NEXT
60 DATA 10, 15, 35, 40, 100, 90, 200, 90, 20, 180
```

In this example the cursor is positioned to location 10, 15 on the first pass. Note that "=X;" takes the place of "10" in the string and "=Y;" takes the place of "15".

If the variable after the "=" is a complex expression (e.g. A+Y), the expression will not be evaluated and the value of the first variable (e.g. A) will be used.

DRIVE

Syntax

DRIVE *expression*

Purpose

This statement causes a new default drive number to be defined. This default drive number is then used by other disk commands when the drive number portion of the file specification is omitted.

Arguments

expression must evaluate to a number between 0 and 3.

Potential Errors

DN - the drive number is either out of range or not specified.

Examples

```
DRIVE A

DRIVE 2
```

Notes/Suggestions

Disk BASIC does not provide a way to determine what the default value is currently set to. Obviously, if you never leave the direct command mode, you will always know what the default is, since BASIC initializes the default to drive #0 on power-up, and the only time it can change is when you give the computer an explicit command to do so. But within a program, it is a different story, since the program can never be sure that the default drive is set to any specific value. Fortunately, there is an easy way to overcome this small problem. The default drive number is stored at memory location &H095A--it is a simple case of PEEKing into this location to determine what value is stored there. (POKEing a number from 0 to 3 into this address has exactly the same effect as executing a DRIVE instruction.)

DSKINI

Syntax

DSKINI *drive [, skip factor]*

Purpose

This statement is used for formatting the diskette in the specified *drive*, with the specified *skip factor*, which determines the order in which sectors are physically written to the disk--if not specified, BASIC uses a default value of 4. Any BASIC program resident in memory at the time the command is executed will be lost. When complete, the routine reads back each sector on each track to verify that it can in fact be read. The final result is a diskette formatted to have 35 tracks of 18 sectors each, where each sector contains 256 bytes of &HFF.

Arguments

drive may be any expression which evaluates to a number between 0 and 3.

skip factor must evaluate to a number between 0 and 16.

Potential Errors

DN - *drive* is either out of range or unspecified.

IO - the disk is not inserted correctly; the drive door is not closed; there is a bad connection between the computer and the drive; the drive itself is faulty; the disk did not format correctly, resulting in bad reads; or the disk has a physical flaw on it.

WP - the disk has a "write protect" tab on it.

Examples

```
DSKINI 0

DSKINI 1, 8
```

Notes/Suggestions

The *skip factor*, when specified, is used by BASIC in determining the physical order of sectors on each track at the time the disk is formatted. If sectors are written on the disk in numerical order (which indicates a *skip factor* of 0), this allows no time for processing between sector reads, which means that the disk has to rotate at least one extra time before the next sector can be read. Each extra disk rotation costs valuable time, especially when a lengthy file is being loaded. To reduce the delay, BASIC sets up a default *skip factor* of 4, which causes the sectors to be written in quite a different order (1, 12, 5, 16, 9, 2, 13, 6, 17, 10, 3, 14, 7, 18, 11, 4, 15, 8). If you consider this to be a circular table, you will notice that between every pair of sequential sectors, there are 4 other sectors which will be "skipped" during the disk rotation. This sequencing seems to provide the optimum disk I/O rate for almost all applications. In our experimentations, we found that *skip factors* less than 4 and higher than 8 produced the most noticeable differences in file loading speed. But of course, our tests were not conclusive, and we invite you to experiment with *skip factor* for yourself.

DSKI$/DSKO$

Syntax

DSKI$ *drive, track, sector, string variable1, string variable2*
DSKO$ *drive, track, sector, string variable1, string variable2*

Purpose

These statements permit direct access to the sectors on a standard formatted diskette. You can input a sector with DSKI$ or output a sector with DSKO$.

Arguments

drive must be a number between 0 and 3.

track must be a number between 0 and 34.

sector must be a number between 1 and 18.

string variable1 and *string variable2* will be defined according to the nature of the operation selected. On input, *string variable1* will contain the first 128 bytes of the sector and *string variable2* will contain the second 128 bytes of the sector. On output, each of these strings must be defined to contain exactly 128 characters of data each.

Potential Errors

FC - *drive*, *track*, or *sector* are out of range

IO - the disk is not properly inserted in the drive; the door is not closed; on a read, it is found that the disk is unreadable; or on a write, it is found that the specified track/sector cannot be found (possibly because the disk is unformatted).

VF - VERIFY is on, and following a write, the controller was unable to read back the same sector.

WP - there is a "write-protect" tab on the disk.

Examples

```
DSKI$ 0, 17, 3, A$, B$

DSKO$ D, T, S, A$(0), A$(1)
```

Notes/Suggestions

Although this may be an example of overstating the obvious, be aware of the fact DSKO$ causes new data to be written to the specified track/sector--that is, any data that exists on that sector at the time you execute the DSKO$ instruction will be destroyed. Therefore, do not use this instruction unless you are completely sure of what you are doing, especially if you are modifying the directory area of the disk (Track 17, Sectors 2 through 11).

EDIT

Syntax

EDIT *line number*

Purpose

This statement allows modification of the specified *line number*. Note that it actually causes you to enter the Edit mode of computer operation. Once you are in the Editor, there are several sub-functions that can be called, which are described below.

Arguments

line number must be any valid line number from 0 to 63999.

Sub-Functions

n SPACEBAR	- Move cursor n spaces to right. If "n" is omitted, 1 is used.
n LEFT ARROW	- Move cursor n spaces to left. If "n" is omitted, 1 is used.
n S *char*	- <S>earch for the nth occurrence of *char*. If "n" is omitted, 1 is used.
n C *new chars*	- <C>hange the next n characters to *new chars*, starting with the character under the cursor. If "n" is omitted, 1 is used.
n D	- <D>elete the next n characters, starting with the character under the cursor. If "n" is omitted, 1 is used.
n K *char*	- <K>ill all characters from the cursor position to the nth occurrence of *char*. If "n" is omitted, 1 is used.
A	- Cancel <A>ll changes, but remain in the Editor.
E	- <E>xit Editor; retain all changes.

Q	- Cancel all changes and <Q>uit the Editor.
L	- Re-display <L>ine, complete with all changes made so far.
I	- Begin <I>nserting new characters, starting at the cursor. All characters under the cursor and to the right will be moved to make room for the new characters.
H	- <H>ack all characters from the cursor to the end of the line, and begin inserting new characters.
X	- E<X>tend line; move the cursor to the end of the line and begin inserting new characters.
SHIFT UP ARROW	- terminate insertion of new characters.
ENTER	- global exit from Editor, regardless of sub-function being executed.

Potential Errors

FC - *line number* is either out of range or unspecified.

UL - *line number* does not exist in memory.

Examples

```
EDIT 1000
```

Notes/Suggestions

During the development phase of any BASIC program, it is sometimes useful to insert a line containing an EDIT instruction into the body of the program. When this is done, program execution stops at the EDIT instruction, and the specified line is displayed for editing. This can be a very helpful tool, especially when you are trying to do something like creating a properly formatted PRINT statement. Of course once the desired format is obtained, the line containing the EDIT command should be removed.

ELSE

SEE *IF*.

END

Syntax

```
END
```

Purpose

This statement is used as a program terminator, and is similar to the STOP statement, except that following an END command, program continuation is not possible. It is not strictly required within the body of the program unless the program must be terminated somewhere in the middle (but its presence does ensure that any OPEN files are properly CLOSEd). That is, if no END statement is encountered, program execution will continue until the physical end-of-program is reached.

Arguments

None.

Potential Errors

None.

Examples

```
IF A = B THEN ... ELSE END
```

EOF

Syntax

numeric variable = EOF *(buffer)*

Purpose

This function is used in conjunction with cassette or disk files, opened for input, to determine whether an End-Of-File condition has occurred. Use of this command helps to prevent attempts to read past the end of the open file. It is only required when the physical size of the file is not known. This is a strictly Boolean (logical) operation; that is, *numeric variable* is given a Boolean value of either -1 for "true" or 0 for "false", where "true" indicates that the end of the file has been reached.

Arguments

buffer may be an expression, but must evaluate to a valid number as follows:

-1	-	for cassette files
1 to 15	-	for disk files

Potential Errors

DN - *buffer* is out of range.

NO - the file that uses *buffer* has not been opened.

Examples

```
A = EOF (2)

IF EOF (DV) = -1 THEN ...

IF EOF (DV) THEN ...
```

Notes/Suggestions

Because of the Boolean nature of the EOF instruction, no equality operator is necessary in an IF...THEN statement. Thus, examples 2 and 3 above perform exactly the same function. Note, however, that example 3 requires less memory and is marginally faster than example 2 (faster because the equality relationship does not have to be evaluated).

EXEC

Syntax

EXEC *[address]*

Purpose

This statement is used for transferring control to a machine-language program or subroutine, where *address* must be the start of a valid machine-language instruction. If control is transferred temporarily to a subroutine which is being used in conjunction with a BASIC program, control can be returned to the BASIC program by means of an "RTS" operation code.

Arguments

address must evaluate to a valid number between &H0000 and &HFFFF (0 and 65535).

Potential Errors

FC - either no *address* has been defined (by a CLOADM or a LOADM or by a previous EXEC command) or the specified *address* is out of range.

Examples

EXEC

EXEC &HA928

Notes/Suggestions

EXEC should be used in lieu of USR for all subroutines which do not involve the passing of parameters to or from the routine. There are two reasons for this suggestion: first, EXEC is faster, does not require as much memory, and generally does not require any external definitions to be made; second, the USR functions are still available for additional subroutines which involve parameters.

EXP

Syntax

numeric variable = EXP *(expression)*

Purpose

This function calculates the value of the Napierian base (2.71828183) raised to the power *expression*. It is the logical inverse of the LOG function (i.e. LOG (EXP (*expression*)) = *expression*).

Arguments

expression may evaluate to any negative number from -10^{-39} to -10^{+37} or any positive number from 10^{-39} to 88.0296919.

Potential Errors

OV - *expression* is out of range.

Examples

```
A = EXP (X + Y)

IF SGN (EXP (Z)) < 1 THEN ...
```

FIELD

Syntax

FIELD *[#] buffer, size* AS *string variable [, ...]*

Purpose

This statement is used for formatting records in direct access disk files. When executed, each record in the file will have a number of variable *size* fields. The number of fields is limited only by the physical limits imposed on the length of a normal BASIC program line.

Arguments

buffer may be any disk buffer number, from 1 to 15, which has been opened for <R>andom or <D>irect access.

size may be any integer number from 0 to 255, as long as the sum of all *size*'s and each individual *size* do not exceed the total record length as specified in the OPEN statement.

Potential Errors

FC - *size* is out of range.

FO - the sum of 1 or more *sizes* exceeds the total record length.

Examples

```
FIELD #1, 15 AS R1$, 15 AS R2$

FIELD 1, R1 AS R1$, R2 AS R2$, R3 AS R4$
```

Notes/Suggestions

When a string variable has been FIELDed for use with GET or PUT, it should not be used for any other purpose because the integrity of the disk file may be destroyed. What's worse is that such a logic error is impossible to spot unless an "?SE ERROR" occurs following an LSET or RSET attempt. Consider the following broken sequence of BASIC instructions:

```
100 FIELD #1, 10 AS A$, 10 AS B$

     ...program continues...

200 A$ = "HELLO"

     ...program continues...

300 GET #1, R
```

Under normal circumstances, you would expect the GET statement in line 300 to assign values to both A$ and B$, depending on what is contained in the disk file. Unfortunately, this is not the case. Although B$ is indeed properly defined, examination of A$ will reveal that it has been set equal to "HELLO", which is not at all what the program intended. In this example, no error is generated, making it quite difficult to locate the logic bug.

Consider the following different sequence of instructions:

```
100 FIELD #1, 10 AS A$, 10 AS B$

     ...program continues...

200 A$ = "HELLO"

     ...program continues...

300 LSET A$ = STRING$ (10, "*")
```

This time, an "?SE ERROR" occurs in line 300, indicating that the interpreter no longer considers A$ to be fielded (because of the intervening assignment in line 200). This makes the debugging process a little easier, but if you think about it, you will realize that the error is still the same.

The moral of the story is that FIELDed string variables should only be used with GET, PUT, LSET and RSET statements...

FILES

Syntax

FILES *[buffers] [, memory space]*

Purpose

This statement defines the number of disk *buffers* active and/or the amount of additional *memory space* to be reserved for the *buffers*. This memory space is considered additional because BASIC always reserves a minimum of 256 bytes of memory for each buffer. Note that one or both parameters must be present.

Arguments

buffers may be an expression, but must evaluate to a number from 0 to 15.

memory space may be an expression which must evaluate to a number from 0 to the amount of free memory remaining.

Potential Errors

FC - *buffers* is out of range, or *memory space* is greater than &H7FFF (32767).

OM - *memory space* is greater than the amount of free memory remaining.

Examples

```
FILES 0, 0

FILES , 3000

FILES 2
```

Notes/Suggestions

Because of a bug in Disk BASIC ROM Version 1.0, the FILES command does not always work as advertised. Worse yet, the error is inconsistent in that it happens under some circumstances but not under others. The problem can usually be overcome by means of a couple of reverse referencing GOTO's, as follows:

```
00010 GOTO 30
00020 GOTO 40
00030 FILES 3, 256
     :GOTO 20
00040 your program goes here
```

Generally, if you are increasing the number of buffers, the reverse referencing routine should be at the beginning of your program; on the other hand, if you are decreasing the number of buffers, the routine should be near the end of the program.

Unfortunately, this type of fix does not always work (as it does with PCLEAR). If you get an "?SN ERROR" when you run the program, usually it will disappear when you run it a second time. Alternatively, you can remove the FILES statement from the body of the program, and type it in directly from the keyboard before loading the program.

FIX

Syntax

numeric variable = FIX (*expression*)

Purpose

This function truncates the fractional part of *expression* and returns the integer part as the result. It is similar to INT except that INT performs rounding whenever *expression* is negative, whereas FIX does not.

Arguments

expression may be any positive or negative value within the full range from 10^{-38} to 10^{+38}.

Potential Errors

OV - *expression* is out of range.

Examples

```
A = FIX (A)

IF B = FIX (B) THEN ...
```

FN

Syntax

numeric variable = FN *function name (expression)*

Purpose

This function allows you to obtain a numeric result from a function of one or more variables. The function being named must have been previously defined by the DEF FN statement.

Arguments

function name may be any combination of upper case letters and/or digits, as long as the first character is an upper case letter. If the name is more than 2 characters in length, only the first 2 characters will be recognized.

The limits on *expression* will vary according to the nature of the function, but it must always evaluate to a numeric quantity.

Potential Errors

FC - *expression* is out of range for one of the terms of the function.

OV - same as FC.

UF - *function name* has not been defined.

Examples

```
A = FN Z (X * Y)

A = FN Z (FN Z (3))

IF FN Z (A) < A THEN ...
```

Notes/Suggestions

The FN instruction can save a fair amount of memory if you have a function which is to be called from many different places in the program. It is, however, dreadfully slow when compared to the equivalent set of simple in-line statements. In fact, in some cases, we have noted as much as a 50% reduction in speed when the FN statement was replaced by the equivalent in-line code. Obviously, the more complex and/or deeply nested the function, the more time is used up in the calculation. Therefore, unless memory is a serious consideration in your program, we don't recommend the usage of this particular command.

FOR-STEP-NEXT

Syntax

FOR *variable* = *start* TO *end* [STEP *rate*] ... NEXT [*variable*]

Purpose

This composite statement defines a repetitive loop. The set of statements between FOR and NEXT will be executed a predetermined number of times, depending on the magnitude of *start*, *end*, and *rate*. The loop may be of very small size, encompassing no instructions at all, as in a time-wasting loop, or it may span the whole program. Loops may be nested within other loops to a degree that will exceed most application requirements.

Arguments

variable must be any valid numeric variable name. Note that *variable* may be omitted from the NEXT portion of the statement; in this case, when a NEXT is encountered, the computer assumes that the most recently named loop variable is to be used.

start, *end* and *rate* may be expressions which evaluate to positive or negative numbers in the entire range of the computer (i.e. 10^{-38} to 10^{+38}). If *rate* is omitted, a value of +1 is used by default.

Potential Errors

NF - a NEXT was encountered before a loop was properly initiated by means of the FOR statement. (Note that the reverse never occurs--that is, if a loop is started with the FOR statement but the corresponding NEXT is missing, no error will be generated.)

Examples

```
FOR I = 1 TO 10: NEXT

FOR I = 1 TO 10 STEP 2: NEXT I
```

Notes/Suggestions

Although it is quite acceptable to BASIC to exit from a loop without properly terminating it by means of the NEXT statement, this is not considered good programming practice.

Granted, BASIC is able to resolve these incompleted loops most of the time. The hardware stack is never properly cleaned up, however, when you exit by means of a GOSUB to a routine that does not terminate with a corresponding RETURN. What this means is that it is quite possible to cause stack overflow if the incomplete FOR loop is executed enough times, which will result in an "?OM ERROR" that is very difficult to trace. To avoid this condition, it is recommended that you exit the loop correctly. If it is necessary to leave the loop before it has completed by itself, all you have to do is force *variable* to some value higher than *end* (or lower if it is a descending loop) and execute a NEXT statement, as in the following example:

```
00010  FOR I = 1 TO 50
00020  A = X * 2 + 34
00030  IF A > B THEN I = 51
      :NEXT I
      :GOTO 1000
00040  GOSUB 200
      :NEXT I
```

An infinite (non-terminating) loop can be implemented if a STEP rate of zero is specified. This could be used instead of an unconditional GOTO in a subroutine loop. For example, the following subroutine, which uses no line number references, will flash a "?" until a key is pressed:

```
01000  FOR T = 0 TO 1 STEP 0
      :I$ = INKEY$
      :IF I$ = "" THEN PRINT CHR$(8) "?";
      :NEXT
      :ELSE T = 2
      :NEXT
      :RETURN
```

FREE

Syntax

numeric variable = FREE *(drive)*

Purpose

This function returns the number of free granules remaining on the disk in the specified *drive*.

Arguments

drive must evaluate to a number from 0 to 3.

Potential Errors

IO - the disk is not properly inserted in the drive; the drive door is not closed; or the directory track (Track #17) is not readable.

Examples

```
A = FREE (0)

IF FREE (DR) < 2 THEN ...
```

GET

Syntax

GET *(x1, y1) - (x2, y2), array variable, [, G]*

Purpose

This statement is used to save the contents of a graphics rectangle. This data can then be placed in a different location with the PUT statement.

Arguments

X1, X2, Y1, Y2 must all be numeric expressions between 0 and 255. If the expression has a decimal value, the integer portion is used.

variable must be a numeric array variable that has been dimensioned. The following formulas can be used to calcualate the required array size:

1. Using the "G" option,

$$size = \frac{width \times height}{n} - 1$$

where *width* and *height* are the size of the graphics rectangle in pixels and the value of *n* is determined by the graphics mode being used.

PMODE	value of *n*
3 & 4	40
2 & 1	80
0	160

The intermediate result of *width* X *height* should be rounded up to the next whole number, as should the final answer.

2. Without the "G" option,

$$size = \frac{\frac{width}{n} \text{ X } height}{5} - 1$$

where *width* and *height* are the size of the graphics rectangle in pixels and the value of *n* is determined by the graphics mode being used.

PMODE	value of *n*
3 & 4	8
2 & 1	16
0	32

The intermediate result of *width / n* should be rounded up to the next whole number, as should the final answer.

The resulting value of *size* from method 1 or 2 must be used in a DIMension statement to reserve room for the graphics data (eg. DIM GR(34)).

Potential Errors

FC - the size of the rectangle is larger than the array size; the *co-ordinates* are out of range; or *variable* has not been dimensioned.

Examples

```
GET (A, B) - (A+30, B+30), GR, G
GET (20, 20) - (30, 30), DD
```

Notes/Suggestions

When the "G" option is used, the graphics data is stored in complete detail. This ensures smooth animation in graphics games, etc. It also slows down the execution speed of the command. The following two programs were timed to measure the difference in speed.

PROGRAM A

```
00010 PMODE 4
00020 DIM A (255)
00030 FOR T = 1 TO 10
00040 GET (0, 0) - (100, 100), A, G
00050 NEXT
```

PROGRAM B

```
00010 PMODE 4
00020 DIM A (262)
00030 FOR T = 1 TO 10
00040 GET (0, 0) - (100, 100), A
00050 NEXT
```

Program B executed in slightly over 1.1 seconds while program A took 9.6 seconds. Repeated experiments in other PMODEs yielded similar results. Generally, a speed increase of 5 to 8 times can be expected when the "G" option is deleted. A corresponding speed difference is found when the PUT command is used to replace the data on the graphics screen with or without the *action* options.

The following general rules can be used in determining if the "G" option should be used:

1. If any of the PUT *action* options need to used, the "G" must also be used. Note, however, that the effect of PUT without an *action* is similar to that of one using PSET as the *action*.

2. If the origin and destination addresses of the rectangle are different in PMODE 2 or 4, the "G" option must be used. If it is not specified, garbage may result when the data is PUT. Keep this in mind when GET/PUT are used to save a background display during game animation in PMODE 4. You do not need to use "G" in these cases.

3. If very smooth animation in PMODEs 0, 1, or 3 is required, "G" should be used.

When a program using GET and PUT is being developed, routines should be tried without the "G". Then, if the results are unsatisfactory, the "G" can be added. In many cases a combination will be most efficient. Just remember that if PUT *actions* are to be specified, the "G" must be used in GET and if the "G" is used, the PUT *action* must be used.

GET

Syntax

GET #*buffer [, record number*]

Purpose

This statement is used to retrieve data from a direct (random) access disk file and place it in a disk buffer. This data can then be assigned to variables with INPUT and LINE INPUT, or automatically to variables that have been FIELDed.

Arguments

Buffer - must be a numeric expression equal to an OPEN disk file buffer (i.e. between 1 and 15).

Record number - can be any numeric expression between 1 and 32767. If no record number is specified, the current record pointer (LOC) is used. LOC is incremented after every GET.

Potential Errors

BR - *record number* is 0; the record specified cannot exist on the disk due to the physical size of the disk; or the record specified causes the file size to exceed 612 sectors.

DN - the *buffer* specified refers to a disk buffer not reserved with FILES, or *buffer* is out of range but less than 32768.

FC - *record number* is less than 0 or greater than 32767; or *buffer* is less than 1 or greater than 32767.

IE - the record specified is past the end of the file.

IO - the disk is not properly inserted in the drive; the drive door is not closed; or the disk is unreadable.

NO - the *buffer* specified has not been OPENed.

Examples

```
GET #1

GET #8, LOF(8) + 1
```

GOSUB-RETURN/ON-GOSUB-RETURN

Syntax

GOSUB *line number ...* RETURN
ON *expression* GOSUB *line1 [, line2] [, ...] ...* RETURN

Purpose

The GOSUB statement allows you to temporarily pass control to a subroutine starting at the specified *line number*. If used in conjunction with the ON statement, control may be passed subject to the outcome of a conditional test to one of a series of different line numbers. In both cases, the subroutine is terminated by a RETURN statement, which causes the main program to be re-entered at the statement following the GOSUB call.

Arguments

line number, line1, line2, etc., must all be valid numbers between 0 and 63999.

expression must evaluate to an integer number between 1 and 255.

Potential Errors

FC - *expression* is out of range.

UL - the line number referenced by the GOSUB or ON-GOSUB statement does not exist in memory.

Examples

```
GOSUB 1000

ON A GOSUB 100, 200, 300

ON SGN (A) + 2 GOSUB 100, , 200
```

Notes/Suggestions

The ON-GOSUB statement in example 2 above tells the computer "if A = 1 then GOSUB 100; else if A = 2 then GOSUB 200; else if A = 3 then GOSUB 300". But what if A is some value other than 1, 2, or 3? In that case, control is passed to the statement following the ON-GOSUB statement, even if it is on the same line. This, of course, is true only when A is between 1 and 255.

Notice the syntax of example 3, in which a line number has been deliberately omitted from the list. This format is useful when it is known beforehand that the expression will never be equal to the value which references the missing line (in our example, the value is 2.) BASIC treats this as a reference to line #0 and will produce a "?UL ERROR" only when A = 2 and line #0 does not exist in memory. Use of this syntax can save you quite a few bytes of memory, especially when the potential range of values for the expression is large, but only a few of the values will ever be used in the ON-GOSUB statement. There is, however, one major glitch in this type of format: whenever you renumber the program, all line numbers following a blank comma will not be renumbered correctly. For this reason, use the format only in the final, correctly numbered version of your program.

GOTO/ON-GOTO

Syntax

GOTO *line number*
ON *expression* GOTO *line1 [, line2] [, ...]*

Purpose

The GOTO statement allows you to unconditionally pass control to a program segment starting at the specified *line number*. If used in conjunction with the ON statement, control may be passed, subject to the outcome of a conditional test, to one of a series of different line numbers.

Arguments

line number, line1, line2, etc., must all be valid numbers between 0 and 63999.

expression must evaluate to an integer number between 1 and 255.

Potential Errors

FC - *expression* is out of range.

UL - the line number referenced by the GOTO or ON-GOTO statement does not exist in memory.

Examples

```
GOTO 1000

ON A GOTO 100, 200, 300

ON SGN (A) + 2 GOTO 100, , 200
```

Notes/Suggestions

The ON-GOTO statement in example 2 above tells the computer "if A = 1 then GOTO 100; else if A = 2 then GOTO 200; else if A = 3 then GOTO 300". But if A is some value other than 1, 2, or 3 (within the range 1 to 255), then control is passed to the statement following the ON-GOTO statement, even if it is on the same program line.

In example 3, we have shown that it is possible to have a list of line numbers with "holes" in it. This format is useful when you know beforehand that the expression will never be equal to a value which will cause a missing line number to be referenced. BASIC will treat the missing line number as a reference to line #0. In our example, BASIC will report a "?UL ERROR" only if the expression is equal to 2 AND line #0 does not exist in the program. This syntax can save you several bytes of memory space when the potential range of values for *expression* is large but only a few of them will ever be used to reference existing lines. But beware! If you try to renumber a program containing this type of line, all line numbers following a blank comma will be left as they are. Therefore, use this format with caution, and then only in the final version of your program.

HEX$

Syntax

string variable = HEX$ *(expression)*

Purpose

This function converts *expression* from a numeric decimal quantity to an ASCII string of from 1 to 4 hexadecimal digits.

Arguments

expression must evaluate to a number from 0 to 65535.

Potential Errors

FC - *expression* is out of range.

Examples

```
A$ = HEX$ (1000)

PRINT HEX$ (A)

A$ = RIGHT$ (("000" + HEX$ (A)), 4)
```

Notes/Suggestions

Example 3 above provides a quick and convenient method of obtaining a formatted 4-character hexadecimal string.

Conversion of a hexadecimal string back to decimal can be accomplished by means of the *VAL* function in conjunction with "&H" notation, as in A = VAL ("&H" + HEX$ (A)).

IF-THEN-ELSE

Syntax

IF *condition* THEN *statement1* [ELSE *statement2*]
IF *condition* GOTO *line number* [ELSE *statement2*]
IF *condition* GOSUB *line number* [ELSE *statement2*]

This composite statement allows for conditional execution of segments of BASIC code. Whenever *condition* is found to be true, *statement1* is executed or control is passed to *line number*. If *condition* is found to be false then BASIC checks for the existence of an ELSE statement. If one exists, *statement2* is executed; otherwise, control is passed to the line following the IF-THEN statement.

Lines 2 and 3 of the syntax lines above are given to show that the THEN statement may be omitted only when it would be followed by a GOTO or GOSUB statement.

Arguments

condition may be any expression involving the use of optional relational operators (<, =, or >) that will be evaluated by BASIC to a value of -1 for true or 0 for false.

statement1 and *statement2* may be any legitimate BASIC command or combination of commands separated by colons. As well, they may be line numbers, which will be treated by BASIC as if they were preceded by GOTO statements.

line number may be any number from 0 to 63999.

Potential Errors

The types of errors possible will be determined by the types of commands used in the arguments list.

Examples

```
IF A = -1 THEN 200

IF A < 0 THEN PRINT "DONE": END ELSE 200

IF A + B <> 15 THEN A = 1: B = 1: GOTO 200

IF A GOSUB 300: GOTO 200 ELSE A = NOT (A)
```

Notes/Suggestions

Relational evaluations are not restricted to use within IF-THEN statements. They can be employed anywhere, even directly from the keyboard. For example, typing "PRINT A < 0" will cause the computer to respond with either -1 or 0 depending on the current value of A. Similarly, instead of using the PRINT statement, you could just as easily assign the result of such a test to another variable, as in "B = A < 0". In this case, B will now take on the value -1 or 0, depending on the current value of A. Now that B has been given a Boolean (logical) value, the IF statement can be used to test B directly, without the need for any relational operators, as in "IF B THEN ...".

An understanding of this technique can sometimes totally eliminate IF-THEN-ELSE statements. For example, suppose you wanted to toggle a variable between 13 and 19 each time through a program segment. Traditionally, this would be accomplished by means of a statement like

```
IF A = 13 THEN A = 19 ELSE A = 13.
```

But, considering that 13 and 19 could just as easily have been -1 and 0, it is possible to rewrite the line without the use of the IF-THEN-ELSE statement, as follows:

```
A = -6 * (A = 13) + 13
```

Now study this statement closely. The relation in parentheses will always result in a value of -1 or 0. Multiplying that value by -6 will result in a value of 6 or 0. Adding 13 will result in value of 19 or 13. Variable A does not even have to be defined the first time through the loop, in which case it will be given the value of 13. (Remove the parentheses and A will always have the value of 0--think about it.)

What is the advantage of this unusual type of notation? For one thing, you can now append a series of additional statements to the end of the line, without worrying about whether or not they will be executed. This means a saving of a few bytes of memory in its own right. For another thing, the second notation takes up less memory than the first, although, admittedly, it does execute more slowly. We will leave it to you to experiment further with the above variation and to decide for yourself where or when to use it.

INKEY$

Syntax

string variable = INKEY$

Purpose

This function allows you to scan the keyboard for a single key stroke (any key) and to retain the ASCII value of that keystroke in *string variable*.

Arguments

None.

Potential Errors

None.

Examples

```
A$ = INKEY$

IF INKEY$ = "" THEN ...
```

INPUT/LINE INPUT

Syntax

```
INPUT [*"message";*] variable list
INPUT #buffer, variable list
LINE INPUT [*"message";*] string variable
LINE INPUT #buffer, string variable
```

Purpose

These statements allow you to obtain data either from the keyboard or from a cassette or disk file which has been opened for input (a disk file may also be opened for direct or random access). If the data source is the keyboard then the statements allow the printing of a message as part of the command. During execution of the program, the requirement of data input from the keyboard is signified by the presence of BASIC's normal flashing cursor. When this appears, simply type the necessary data. If you make a mistake, you can press the left arrow to erase the last character and retype it. <SHIFT> <LEFT ARROW> will cause the entire line to be erased, and <CLEAR> will cause the screen to be cleared. If multiple entries are to be made, they can be separated by commas. To terminate data entry, press the <ENTER> key.

Arguments

"message" may be any combination of characters enclosed by quotation marks and terminated by a semi-colon. The INPUT statement will add a "? " (question mark followed by a space) at the end of the message. The LINE INPUT statement displays the message exactly as printed without adding the extra characters.

buffer may be -1 for a cassette file or 1 to 15 for a disk file.

variable list may be any number of numeric or string variables (simple or array type). If more than one variable is listed, then the second and subsequent variables must be preceded by commas. Note that this format is acceptable only for the INPUT statement.

string variable may be any simple or array string variable. LINE INPUT will allow only one variable name as its argument.

Potential Errors

ER - you have attempted to INPUT some data from a direct (random) access disk file without having first performed a corresponding GET.

FD - you have attempted to read string data from an open file into a numeric variable.

FM - the specified *buffer* has not been opened for input.

ID - you have attemped to use INPUT or LINE INPUT in the direct mode. These commands may only be used within a BASIC program.

IE - you have attempted to read data from past the end of the file. Use EOF to determine if the end of file has been reached.

OS - you have not cleared enough string space.

You may also experience the following messages during INPUT from the keyboard (they do not appear if LINE INPUT is used):

?? - The computer is expecting more than one data item to be entered, but you have not completed the list. The "??" will continue to appear until all specified variables have been assigned a value.

REDO - The computer is expecting numeric input but you have entered non-numeric (string) data.

EXTRA IGNORED - You have entered more data items (separated by commas) than the computer required.

Examples

```
INPUT "ENTER YOUR NAME (LAST, FIRST)"; L$, F$

INPUT #1, A$, B$, C$

LINE INPUT "ENTER YOUR NAME? "; N$

LINE INPUT #-1, A1$
```

Notes/Suggestions

There are some interesting differences between cassette files and disk files, in terms of reading the data by means of the INPUT statement. In cassette files, INPUT will return only 7-bit ASCII data (in fact, the ASCII value of each character in the string will be between &H20 and &H7A), even though the data may have been correctly written in 8-bit format. Furthermore, when you issue a command like "INPUT #-1, A$", BASIC will read data from the buffer until a carriage return is encountered or the length of A$ is 255 characters.

In disk files, the data is read back in full 8-bit binary format, which means that the full range of values from 0 to 255 can be read into a string. As well, input is terminated not only when a carriage return is detected or when the string length reaches 255 characters, but also when a comma is detected.

On the other hand, LINE INPUT treats the data in both cassette files and disk files in exactly the same manner. Individual characters are limited to 7-bit ASCII format in the range &H20 to &H7F, and input continues until string length reaches 255 characters or a carriage return is detected.

Thus, if you wish to write a program that contains a general-purpose routine to read data from an open file, whether it is a cassette file or a disk file, you must use the LINE INPUT statement in order to remain completely general.

When using INPUT or LINE INPUT to read data from a direct access disk file, you must use the GET statement before attempting input, as in the following example:

```
00010 OPEN "D", #1, "FILE.DAT:1"
00020 EF = LOF (1)
      :R = 0
00030 R = R + 1
      :IF R > EF THEN CLOSE #1
      :END
00040 GET #1, R
      :INPUT #1, A$
      :PRINT A$
      :GOTO 30
```

INSTR

Syntax

numeric variable = INSTR *([position,] search string, target)*

Purpose

This function allows you to examine a *search string* for an occurrence of *target*. If specified, *position* tells the computer at which character to begin the search. If it is not specified, the search will always start at the first character in *search string*. If *target* is found in *search string*, the position of *target* is returned. If *target* does not exist, a value of 0 is returned.

Arguments

position may be any value between 1 and 255.

search string and *target* may be any legitimate strings of from 0 to 255 characters in length, and each character in the strings may have any value between 0 and 255.

Potential Errors

FC - *position* is out of range.

Examples

```
A = INSTR (A$, "THE")

A = INSTR (P, A$, "THE")

IF INSTR ("123456", A$) <> 0 THEN ...
```

Notes/Suggestions

The INSTR function, in conjunction with the INKEY$ function, provides a very convenient method of allowing a user to select from a list of menu options, as in the following example:

```
00100 A$ = INKEY$
00110 IF A$ = "" THEN 100
00120 ON INSTR ("123", A$) GOTO 200, 300, 400
00130 GOTO 100
```

INT

Syntax

numeric variable = INT *(expression)*

Purpose

This function converts a rational number to an integer. When *expression* is positive, the result is exactly the same as for the FIX function--i.e. the fractional part of *expression* is merely truncated, and the integer part is retained. When *expression* is negative, however, the result is a little bit different. In this case, INT causes the magnitude of the integer portion of the number to be increased by one if the fractional part is not precisely equal to zero. For example, INT (-3.0000001) = -4.

Arguments

expression may evaluate to any positive or negative number from 10-38 to 10+38.

Potential Errors

OV - *expression* is out of range.

Examples

```
A = INT (X * Y + Z)

IF A <> INT (A) THEN ...
```

JOYSTK

Syntax

numeric variable = JOYSTK *(potentiometer)*

Purpose

This function allows you obtain the analog value of one of the four joystick potentiometers. The value returned is always between 0 and 63.

Arguments

potentiometer must evaluate to a number between 0 and 3. These values correspond to the following joystick readings:

0 - right joystick horizontal data
1 - right joystick vertical data
2 - left joystick horizontal data
3 - left joystick vertical data

Potential Errors

FC - *potentiometer* is out of range.

Examples

```
A = JOYSTK (0)

SET (JOYSTK (0), JOYSTK (1) / 2, 4)

IF JOYSTK (3) > 31 THEN ...
```

Notes/Suggestions

To read the status of the joystick buttons you must PEEK memory location &HFF00 (65280). If the value at this memory location is ANDed with 3 (To assign the values to variable "A" you could use A = (PEEK (&HFF00) AND 3)) a number of unique values will be returned as follows:

3 - no buttons are pressed
2 - right button is pressed
1 - left button is pressed
0 - both buttons are pressed

KILL

Syntax

KILL *file specification*

Purpose

This statement is used for deleting files from a disk, making the space available for re-use by other files.

Arguments

file specification consists of 3 parts; *filename* and *extension* are mandatory, but *drive* is optional:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (","), slash ("/"), or colon (":").

extension - any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/").

drive - must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

Potential Errors

FN - illegal *file specification*.

IO - the disk is not properly inserted in the drive; the drive door is not closed; or a write error occured on the directory track.

NE - *file specification* does not exist on the disk.

Examples

```
KILL F$

KILL "PROGRAM/BAS:0"
```

LEFT$

Syntax

string variable = LEFT$ *(string expression, length)*

Purpose

This function allows you to obtain a portion of *string expression*, starting with the leftmost character, consisting of the number of characters specified by *length*. If *length* is greater than the total length of *string expression*, then the entire string is returned.

Arguments

string expression may be any combination of from 0 to 255 ASCII characters; each character may have any ASCII value between 0 and 255.

length may be an expression which must evaluate to a number between 0 and 255.

Potential Errors

FC - *length* is out of range.

Examples

```
A$ = LEFT$ (B$ + STRING$ (22, B) + C$, 32)

IF LEFT$ (A$, 1) = "Y" THEN ...
```

LEN

Syntax

numeric variable = LEN (*string expression*)

Purpose

This function returns the length of *string expression*, which will always be a value between 0 and 255.

Arguments

string expression may be any combination of from 0 to 255 ASCII characters; each character may have any ASCII value between 0 and 255.

Potential Errors

None.

Examples

```
A = LEN (B$ + STRING$ (22, B) + C$)

IF LEN (A$) <> 5 THEN ...

PRINT LEN (A$)
```

LET

Syntax

LET *variable name = expression*

Purpose

This statement is a rarely used instruction which allows you to assign a value (either numeric or string) to a corresponding variable type.

Arguments

variable name may be any combination of upper-case letters and decimal digits, as long as the first character in the name is an upper-case letter. If the length of the name exceeds 2 characters, only the first 2 characters will be recognized by BASIC. If a "$" follows the variable name, that variable will be treated as a string variable; otherwise, it will be treated as a numeric variable.

expression may be either a numeric expression, which must evaluate to a positive or negative number between 10^{-38} and 10^{+38}, or a string expression which must evaluate to a string of from 0 to 255 ASCII characters, each of which may have any ASCII value from 0 to 255.

Potential Errors

None.

Examples

```
LET A$ = "NOW IS THE TIME ..."

LET A = 1.23 * X + 55
```

Notes/Suggestions

LET is provided as an optional statement to make Color BASIC more compatible with other forms of BASIC which require its presence. Logically, it makes sense to use this statement in assigning values to variables, since it helps to remove the ambiguity associated with a statement like "A = 1.23 * X + 55". It is not practical, however, to use the LET statement for each variable assignment, because each occurrence uses up an additional byte of program memory. This may not seem like much until you consider that it is quite easy to have 500 or more variable assignments in a single program--that's 500 bytes of valuable memory which could be used for another one or two subroutines.

LINE

Syntax

LINE *[(X1, Y1)] - (X2, Y2), mode [, B[F]]*

Purpose

This statement draws a single line on the high-resolution graphic screen from the starting co-ordinate to the ending co-ordinate. If no starting co-ordinate *(X1, Y1)* is specified, the line is drawn from the last named co-ordinate, which BASIC initializes to (128, 96). If used, the B option causes a rectangle (box) to be drawn, in which case the specified co-ordinates define the upper left and lower right corners of the box. If additionally the F option is specified, the box will be color-filled; the color chosen will be the foreground color if *mode* was "PSET" or the background color if *mode* was "PRESET".

Arguments

X1 and *X2* are the horizontal components of the co-ordinates and must evaluate to any number between 0 and 65535, although BASIC will modify the number to between 0 (left side of the screen) and 255 (right side of the screen).

Y1 and *Y2* are the vertical components of the co-ordinates and must evaluate to any number between 0 and 255, although BASIC will modify the number to between 0 (top of the screen) and 191 (bottom of the screen).

mode must be one of the BASIC keywords "PSET" (to draw the line) or "PRESET" (to erase the line).

The B option may appear by itself but the F must be preceded by the *B* option.

Potential Errors

FC - *X1*, *X2*, *Y1*, or *Y2* are out of range.

Examples

```
LINE (0, 0) - (255, 191), PSET

LINE - (X, Y), PSET, B

LINE - (X+33, Y+28), PRESET, BF
```

Notes/Suggestions

You can obtain a wide variety of interesting screen displays by POKEing different values (between 0 and 255) into memory locations &H00B2 (foreground color) and &H00B3 (background color). Of course, you won't notice much difference from normal line displays unless you use the BF option.

The statement arguments listed above are valid regardless of which PMODE is active; BASIC fixes the co-ordinate values so that they are within the proper range.

LINE INPUT

SEE *INPUT/LINE INPUT*

LIST/LLIST

Syntax

LIST *[startline] [-] [endline]*

LLIST *[startline] [-] [endline]*

Purpose

This statement causes the BASIC program in memory to be LISTed to the screen or LLISTed to the printer. The command may include a *startline*; if none is specified, the first program line in memory is used. It may also include an *endline*; if none is specified, the listing continues until the end of the program is reached.

Arguments

Both *startline* and *endline* may be any number from 0 to 63999. If *endline* is smaller than *startline*, no listing will be generated.

Potential Errors

None.

Examples

```
LIST

LIST - 200

LLIST 150 -

LLIST 100 - 500
```

Notes/Suggestions

LIST and LLIST may be included within a BASIC program (this may be handy for de-bugging purposes), but as soon as the command is encountered, the listing is generated and then program execution ceases.

LOAD

Syntax

LOAD *file specification [, R]*

Purpose

This statement allows you to load a BASIC program from disk. If the [, R] option is included, OPEN files will remain OPEN and the new program will begin executing automatically. If not, all files are CLOSEd and the program must be executed directly from the keyboard by means of the RUN command.

Arguments

A *file specification* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (","), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extension "BAS".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

Potential Errors

FM - the specified file is not a BASIC program.

IO - the disk is not properly inserted in the drive; the door is not closed; the program is loading into bad memory; or the disk is unreadable.

NE - the specified file cannot be located on the disk.

Examples

```
LOAD "PROGRAM/BAS:1"
LOAD F$, R
```

LOADM

Syntax

LOADM *[file specification] [, offset]*

Purpose

This statement allows you to load a machine-language program into memory. If *offset* is supplied, this value will be added to the start, end and entry addresses of the program, thus causing the program to load into a different place in memory. Once loaded, the program may be executed by means of the EXEC command. Note that OPEN files are left OPEN when this command is executed.

Arguments

A *file specification* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (".), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extendion "BIN".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

offset may be any numeric expression which evaluates to between &H0000 and &HFFFF (0 and 65535).

Potential Errors

FC - the specified *offset* is out of range.

FM - the specified file is not a machine-language program.

IO - the disk is not properly inserted in the drive; the door is not closed; the program is loading into bad memory; or the disk is unreadable.

NE - the specified file cannot be located on the disk.

Examples

```
LOADM "PROGRAM", &H1234

LOADM F$
```

Notes/Suggestions

LOADM may be used within a BASIC program to load a co-resident machine-language program. If the program loads into high RAM, however, you should protect memory by means of the CLEAR statement before you load the program. Once the program has been loaded, it can be executed either via the EXEC statement, or via a USR call (providing, of course, the entry address has been properly defined by a DEF USR statement).

LOC

Syntax

numeric variable = LOC *(buffer)*

Purpose

This function returns the current record number of an open disk file.

Arguments

buffer must evaluate to a number between 1 and 15, corresponding to the open file being accessed.

Potential Errors

NO - the specified *buffer* has not been opened.

Examples

```
A = LOC (1)

IF LOC (B) < 100 THEN ...
```

Notes/Suggestions

The LOC function may be used with any type of opened disk file, whether it is opened for input, output, or direct access. Normally, however, it is used only with direct access files.

When a file is first opened for direct access, the LOC points to record #1. Each time a GET or PUT statement is executed, LOC is updated to point to the record following that specified by the GET or PUT. Thus, LOC can be used to sequentially access the file.

LOF

Syntax

numeric variable = LOF (*buffer*)

Purpose

This function returns the highest numbered record in a direct (random) access disk file.

Arguments

buffer must evaluate to a number between 1 and 15, corresponding to the open file being accessed.

Potential Errors

NO - the specified *buffer* has not been opened.

Examples

```
A = LOF (1)

GET #1, LOF (1)
```

Notes/Suggestions

Like LOC, the LOF function may be used with any type of opened disk file, whether it is opened for input, output, or direct access. Normally, however, it is used only with direct access files. When such a file is opened for the first time, LOF is given a value of 0 to start.

Example #2 above not only GETs the last record in the file (this will produce an ?FC ERROR if the file has been opened for the first time), but it also updates the LOC pointer to one record beyond. This provides a quick and convenient method of setting pointers for the purpose of adding data to a file.

LOG

Syntax

numeric variable = LOG *(expression)*

Purpose

This function returns the natural or Napierian (base *e* = 2.71828183) logarithm of *expression*.

Arguments

expression must evaluate to a positive number in the range 10^{-38} to 10^{+38}.

Potential Errors

FC - *expression* is not a positive number.

OV - *expression* is out of range.

Examples

```
A = LOG (X)

A = LOG (SIN (X))

IF LOG (X) < 0 THEN ...
```

Notes/Suggestions

BASIC does not provide a means of obtaining the common or base 10 logarithm of a number. A simple conversion factor allows you to find the common logarithm quite easily, as follows:

```
A = LOG (X) / LOG (10).
```

Since "LOG (10)" is a constant, the statement can be rewritten:

```
A = LOG (X) / 2.30258509.
```

LOGICAL OPERATORS

Syntax

numeric variable = *expression* AND *mask*
numeric variable = *expression* OR *mask*
numeric variable = NOT *expression*

Purpose

Logical operators are used for modifying the bit patterns contained in *expression*. AND and OR use another bit pattern, called a *mask* to modify the data, while NOT inverts (or complements) each bit in the data pattern. These statements will operate on 16 bits of data.

Arguments

Both *expression* and *mask* must evaluate to numbers between 0 and 32767 or -1 and -32768.

Potential Errors

FC - *expression* or *mask* is out of range.

Examples

```
A = A AND B OR C

A = A AND (B OR C)

IF NOT (A OR B) THEN ...
```

Notes/Suggestions

The normal order of precedence for logical operations is NOT, AND, OR. This order can be altered by means of parentheses. In example #1 above, BASIC will determine the result of "A AND B" first, and the result will be ORed with C. In example #2, however, the parentheses force the OR operation to be carried out first; the result is then used as a mask for A.

Whenever the result of a logical operation is negative, it can be made positive by adding 65536 to it. If the magnitude of the number was originally below 256, then you can add 256 instead. Now the data can be safely converted to hexadecimal notation, which makes more sense than decimal notation when dealing with bit-patterns.

BASIC does not provide an exclusive-OR statement with its logical operators. This can be artificially performed, however, with the following sequence of instructions, where "V" is the original data value, and "M" is the mask:

```
A = (V OR M) AND (NOT (V AND M)).
```

The use of OR and AND is very common in IF - THEN - ELSE statements. In this case the variables are evaluted to boolean values and the statement is executed in accordance with the truth values returned.

LSET

Syntax

LSET *fielded string variable = string expression*

Purpose

This statement is used in conjunction with direct (random) access disk files. It transfers and left justifies *string expression* into *fielded string variable*. If the length of *string expression* is greater than the field size for the specified variable, then the excess characters are truncated. If the length of *string expression* is less than the field size for the specified variable, then the new string is blank-filled on the right. The primary purpose of this statement is to prevent field overflow when writing new records to the file.

Arguments

string expression must evaluate to any combination of from 0 to 255 ASCII characters, each of which may have any ASCII value from 0 to 255.

fielded string variable may be any legitimate string variable name which has been previously defined in a *FIELD* statement.

Potential Errors

SE - you have named a string variable which has not been previously fielded.

Examples

```
LSET A$ = B$

LSET A$ = "NOW IS THE TIME ..."

LSET A$ = MKN$ (A)
```

Notes/Suggestions

See *FIELD* for additional information about fielded variables.

MEM

Syntax

numeric variable = MEM

Purpose

This function returns the amount of free memory remaining. The result will depend on whether or not there is a BASIC program in memory and how long the program is.

Arguments

None.

Potential Errors

None.

Examples

```
PRINT MEM

IF MEM < 2000 THEN ...
```

MERGE

Syntax

MERGE *file specification [, R]*

Purpose

This statement is used to overlay one BASIC program with another. The program to be merged must be stored on disk in ASCII format, which is done by using the ", A" option when saving the program. When MERGEing, duplicate lines will be overwritten. For example, if the program being merged contains lines 10 and 20, and the program in memory also contains these lines, the resulting program will have line 10 and 20 from the new program and the original lines will be lost. If the [, R] option is used, all OPEN files will remain OPEN and the new program will begin executing automatically, starting at the first program line; otherwise, all files will be CLOSEd.

Arguments

A *file specification* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (".."), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extension "BAS".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

Potential Errors

FM - the specified file is not a BASIC program stored in ASCII format.

IO - the disk is not properly inserted in the drive; the door is not closed; the program is loading into bad memory; or the disk is unreadable.

NE - the specified file cannot be located on the disk.

Examples

```
MERGE "EXAMPLE.PRG"

MERGE F$, R
```

MID$

Syntax

string variable = MID$ (*string, start [, count]*)

Purpose

This function sets a *string variable* equal to a portion of another string. The *start* specifies the first character of *string* to be used and *count* defines the number of characters to be used. If *count* is not specified, then that portion of *string* from *start* to the right end of the string will be returned.

Arguments

string may be any combination of from 1 to 255 characters, each of which may have any ASCII value from 0 to 255.

Both *start* and *count* may be numeric expressions which must evaluate to numbers between 1 and 255.

If *start* is beyond the last character of *string*, a null string will be returned. If *count* is greater than the length of the string from *start*, the string returned will contain everything from *start* to the end of the string.

Potential Errors

FC - *start* or *count* is out of range.

Examples

```
A$ = MID$ (A$, A, 2)

A$ = MID$(A$, 3)

IF MID$ (A$, 3, 2) = "QQ" THEN ...
```

MID$ =

Syntax

MID$ (*string1, start [, count]*) = *string2*

Purpose

This alternate form of the MID$ function allows you to change a portion of a pre-defined *string1*. Note that, in this case, the MID$ statement appears on the left side of the equals sign. When the function is used in this way, *string1* is modified to contain *string2*. If *count* is not specified, then BASIC will use a default *count* equal to the length of *string2*, which is treated as a maximum number of characters to transfer into *string1*; however, BASIC will only continue to transfer characters as long as the original length of *string1* is not changed. For example, if A$ = "HELLO", then the statement MID$ (A$, 3) = "GOODBYE" will result in the new string A$ = "HEGOO".

Arguments

string1 must be a pre-defined string variable.

string2 may be any combination of from 1 to 255 characters, each of which may have any ASCII value from 0 to 255.

Start may be a numeric expression which must evaluate to a number between 1 and 255 and less than or equal to the length of *string1*.

Count may be a numeric expression which must evaluate to between 1 and 255.

If you specify a decimal expression which contains a fractional part (e.g. 3.98), only the integer part of the number (in this case, "3") will actually be used. If *start* is beyond the length of *string1*, no changes will occur. If *count* is greater than the length of the *string1* from *start*, *string1* will be modified to its end.

Potential Errors

FC - *start* or *count* is out of range; or *start* is greater than the length of *string1*.

Examples

```
MID$ (A$, 3, 2) = "**"

MID$ (A$, 3) = "GOODBYE"
```

MKN$

Syntax

string variable = MKN$ *(expression)*

Purpose

This function changes *expression* into a 5-byte string which corresponds to the normal floating-point format for all numeric information. Although used primarily for storing numeric data in a direct (random) access disk file, it can be used anywhere within your program. The string can be converted back to a number by means of the CVN function.

Arguments

string variable may be any legitimate string variable.

expression must evaluate to a positive or negative number in the range 10^{-38} to 10^{+38}.

Potential Errors

OV - *expression* is out of range.

TM - *expression* is not numeric, or *string variable* is not a legitimate string variable.

Examples

```
A$ = MKN$ (234)

IF MKN$ (A) <> 5.9 THEN ...
```

MOTOR

Syntax

MOTOR *argument*

Purpose

This statement causes the cassette motor to be turned on or off. This can be used in conjuction with AUDIO to add spoken instructions or music to a BASIC program. Once turned on, the cassette motor will remain on until a MOTOR OFF instruction is encountered, any BASIC error occurs, or when a CSAVE or CLOAD has been completed.

Arguments

argument must be either ON or OFF.

Potential Errors

None.

Examples

```
MOTOR ON

MOTOR OFF
```

NEW

Syntax

NEW

Purpose

This statement deletes the BASIC program currently in memory and cancels all variable values. It has no effect on USR, EXEC, PCLEAR or CLEAR values.

Arguments

None.

Potential Errors

None.

Examples

NEW

NEXT

SEE *FOR-STEP-NEXT*

NOT

SEE *LOGICAL OPERATORS*

OFF/ON

SEE *AUDIO*
SEE *MOTOR*
SEE *ON-GOSUB-RETURN*
SEE *ON-GOTO*
SEE *VERIFY*

OPEN

Syntax

OPEN *mode, [#] buffer, filespec [, size]*

Purpose

This statement OPENs a specified *filespec* for input, output, or direct (random) access, with communications channelled through *buffer*. Files on tape or disk must be OPENed before they can be written to or read from. When OPENing a disk file, you can include an optional *size* argument, which specifies the the number of bytes to be occupied by each record in the file. If not specified, *size* is defaulted to a value of 256.

Arguments

mode must be either a string variable or a string literal which evaluates to one of the following:

I - to input data from a sequential disk or cassette file.
O - to output data to a sequential disk or cassette file.
R or D - to input or output data to a random access disk file.

buffer must be one of the following (note that it is not necessary to OPEN either the printer or the screen/keyboard):

-2 - printer
-1 - cassette
0 - screen/keyboard
1 to 15 - disk

A cassette *filespec* consists of one part:

filename - any character string of from 0 to 255 characters in length; the ASCII value of each character must be between 0 and 255. If the length of the *filename* is 0, then BASIC will use a default name consisting of 8 spaces. If the length exceeds 8 characters, then only the first 8 characters will be used.

A disk *filespec* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (","), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extension "DAT".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the *filespec* by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

size may be an expression which must evaluate to an integer number between 1 and 32767.

Potential Errors

DF - the OPEN statement resulted in the creation of a new file, at which time the disk was found to be full.

DN - *buffer* is out of range.

FC - *size* is out of range.

FM - *mode* is not legal; or, in a cassette system, the file found is not a data file.

FN - *filespec* is not a legal disk file specification.

IO - the disk is not properly inserted in the drive; the door is not closed; or (opening for input or direct access) the directory track is unreadable.

NE - (opening for input) *filespec* does not exist on the disk.

OB - there is not enough memory space available for use as a buffer (usually occurs when *size* is too large.)

VF - (opening for output or direct access) VERIFY is on; during creation of a new file, the controller was not able to read back a sector on the directory track.

WP - (opening for output or direct access) during creation of a new file, the disk was found to be write protected.

Examples

```
OPEN "I", #-1, "DATA"

OPEN "O", 2, "DATA.DAT"

OPEN "R", 8, "DATA:2", 355

OPEN A$, #3, B$
```

OR

SEE *LOGICAL OPERATORS*

PAINT

Syntax

PAINT *(X, Y) [, paint color] [, border color]*

Purpose

This statement PAINTs a specified area of the graphic screen a specified *paint color*. The area to be painted must be surrounded by an unbroken line of the *border color*, where *(X, Y)* is the co-ordinate pair of a point within that area. If *paint color* is omitted, the current foreground color is used; if *border color* is omitted, the current background color is used.

Arguments

X may be a numeric expression which must evaluate to a number between 0 and 65535, although BASIC will modify the number to between 0 and 255.

Y may be a numeric expression which must evaluate to between 0 and 255, although BASIC will modify the number to between 0 and 191.

paint color may be a numeric expression which must evaluate to between 0 and 8.

border color may be a numeric expression which must evaluate to between 0 and 8.

For any of these arguments, if rational numbers are supplied, only the integer part of the number will be used by BASIC. For example, a *paint color* of 3.652 will cause an actual color value of 3 to be used.

Potential Errors

FC - one of the arguments is out of range.

Examples

```
PAINT (20, 30), 3, 2

PAINT (3*A, 20+B), ,C1/2

PAINT (X, Y), C
```

Notes/Suggestions

You can generate some very interesting patterns with the PAINT statement by altering the foreground color value, which is stored at memory location &HB2. Try POKEing different values into this location and using the PAINT statement without specifying a *paint color*:

```
00010 POKE &HB2, &H4D
00020 PAINT( X, Y ), , 3
```

The above example will PAINT the area starting at X, Y to a striped pattern until a *border color* of 3 is reached. The effect of this "wallpaper poke" varies with different PMODEs.

PCLEAR

Syntax

PCLEAR *expression*

Purpose

This statement reserves memory pages for use in high resolution graphics applications. It also resets all variables in the same manner that CLEAR does. Memory is reserved in blocks of 1536 bytes (called "graphic pages".) A PCLEAR 4 (the default after a cold start) reserves 4 * 1536, or 6144 bytes for graphics.

Arguments

expression must evaluate to a number between 1 and 8.

Potential Errors

FC - *expression* is out of range; or there is not enough free memory remaining to reserve the area specified.

Examples

```
PCLEAR 3
```

Notes/Suggestions

In many applications, there is no need to have any memory reserved for graphic use. Unfortunately, the lowest allowable value that the PCLEAR statement will accept is 1, which still reserves 1536 bytes of memory that could be used by your program. It is possible, however, to execute an artificial "PCLEAR 0", with the following statements:

```
POKE &H19, &H06: NEW  (Non-disk systems)
POKE &H19, &H0E: NEW  (Disk systems)
```

Obviously, because of the NEW statement, this method must necessarily be completed before the program is loaded into memory.

When the PCLEAR statement is included in a BASIC program, it is possible that you will experience an "?SN ERROR" when you run the program. Although this may be a legitimate error in the spelling of the keyword, it is more likely that your program is syntactically correct, but has just been subjected to one of the

rare bugs found in Extended BASIC Version 1.0. (Version 1.1 has been fixed to eliminate this problem.) This bug only shows up when your program contains a PCLEAR statement that causes a new PCLEAR value to be selected. In order to overcome this problem, you can do one of three things:

1. Try RUNning the program a second time.
2. Execute the PCLEAR statement directly from the keyboard, and eliminate the statement from the program.
3. Set up the PCLEAR statement within a short program segment that contains at least one reverse-referencing GOTO statement.

Methods 1 and 2 are inconvenient, since they involve more typing than is normally required when RUNning a program. Method 3 is also a minor inconvenience, but at least the fix can be included within your program, as in the following example:

```
00010 GOTO 30
00020 GOTO 40
00030 PMODE 0: PCLEAR 8: GOTO 20
00040 your program goes here
```

If you are using PCLEAR to reduce the number of graphic pages being reserved (e.g. changing from PCLEAR 4 to PCLEAR 2), this program segment should be somewhere near the end of your program. If you are increasing the number of graphic pages (e.g. changing from PCLEAR 4 to PCLEAR 8), the segment should appear near the beginning of your program.

PCLS

Syntax

PCLS *[color]*

Purpose

This statement causes the graphics screen to be cleared to the color specified. If *color* is omitted, BASIC uses the current background color.

Arguments

color may be a numeric expression which must evaluate to between 0 and 8.

Potential Errors

FC - *color* is out of range.

Examples

```
PCLS

PCLS X+1
```

Notes/Suggestions

The syntax of this command allows you to enclose *color* within parentheses, but this is not required and should be avoided, since each parenthesis uses up an additional byte of memory space.

If you wish to experiment with unusual patterns, try POKEing memory location &HB3 (which contains the background color value) with different values between 0 and 255, and then executing the PCLS statement without specifying a *color*.

```
10 POKE &HB3, &H4D
20 PCLS
```

The above example fills the graphics screen with a striped pattern. The effects obtained by using this method will be different for various PMODEs.

PCOPY

Syntax

PCOPY *source page* TO *destination page*

Purpose

This statement causes the contents of *source page* in the graphics area to be copied into the *destination page*. Page size is always 1536 bytes. Since the statement transfers data at such a rapid rate, it can be used to simulate high-speed animation.

Arguments

Both *source page* and *destination page* may be expressions which must evaluate to numbers between 1 and 39.

Potential Errors

FC - *source page* or *destination page* is out of range.

Examples

PCOPY 1 TO 3

Notes/Suggestions

Extreme caution must be exercised when using this command, because BASIC does not check to see if either argument refers to a reserved graphics area. For this reason, if a PCLEAR 4 was executed, followed shortly thereafter by a PCOPY 1 to 5, the BASIC program in memory would be (partially) destroyed, since it would be overwritten by the contents of graphics page 1.

PEEK

Syntax

numeric variable = PEEK *(expression)*

Purpose

This function is used to determine the contents of a location in memory.

Arguments

expression must evaluate to an integer number between 0 and 65535.

Potential Errors

FC - *expression* is out of range.

Examples

```
A = PEEK (X)

IF PEEK (&HC00A) = 0 THEN ...
```

PLAY

Syntax

PLAY *string*

Purpose

This statement is used in the creation of single note music. By varying the tempo, speed, volume, length and pitch, you can produce different melodies as well as interesting sound effects.

Arguments

string may be any combination of the following parameters, which may be listed in any order:

Tempo - "T" followed by a digit between 1 and 255. The higher the value, the faster the melody plays.

Note length - "L" followed by a digit between 1 and 255. The higher the value, the shorter the note length.

Octave - "O" followed by a digit between 1 and 5.

Volume - "V" followed by a digit between 1 and 255.

Pause - "P" followed by a digit between 1 and 255.

Note - "N" followed by a digit between 1 and 12,

or a digit between 1 and 12 followed by a semicolon (;),

or the letters A, B, C, D, E, F or G. These letters represent musical notes and can be made "sharp" by following them with a "#" or "+" and made flat by following them with a "-". Note that both "B+" and "C-" are illegal syntaxes, even though "E+" and "F-" are allowed.

Tempo, volume and note length can be followed by the following symbols:

"+" - adds 1 to the current value.

"-" - subtracts 1 from the current value.

">" - multiplies the current value by 2.

"<" - divides the current value by 2.

If the digit following note length or pause is followed by a dot (.), the value of the note will be increased by 1/2. An additional dot will increase the new value by 1/4 the original value. Any number of dots can be used.

To pass a numeric parameter to *string*, the following syntax is allowed:

operation = variable;

where *operation* is the letter T, V, L, N, P or O and *variable* is any numeric variable. Note that a semicolon must follow *variable*, even if it is at the end of a string.

Substrings can be executed within a PLAY statement with the following:

Xsubstring;

where *substring* is a previously defined string variable. Note that a semicolon must follow the variable name, even if it is at the end of a string.

Potential Errors

FC - this will occur if there is any error in the syntax of the string.

Examples

```
PLAY "T4;V4;O2;L4;1;2;3;4;5"

PLAY "T+N4N5N6N7N8"

PLAY "T5V8L8ABCD+E+FG"

PLAY "V=B;N=NN;"

PLAY "L32AP4DP4L>EP4L>FP4L>XA$;"
```

Notes/Suggestions

The "=" syntax is not supported in the Radio Shack documentation for Extended Color Basic and deserves a bit more explanation. If you wish to pass a variable to a PLAY command, Going Ahead with Extended Color Basic suggests that you convert the variable to a string as in the following example which plays the entire range of pitches available with PLAY:

```
10 FOR O = 1 TO 5
20 O$ = "O" + STR$(O)
30 PLAY O$
40 FOR N = 1 TO 12
50 N$ = "N" + STR$(N)
60 PLAY N$
70 NEXT N, O
```

A much simpler, and faster, loop can be constructed using the "=" option to pass variable values directly to the string, as shown in the following example which plays the same notes as the program above:

```
10 FOR O = 1 TO 5
20 FOR N = 1 TO 12
30 PLAY "O=O;=N;"
40 NEXT N, O
```

When the "=" sign is encountered in a play string, the value of the variable is subsituted for the "=" and the variable. For example, the first time through the above loop, the string would be interpreted as though it was "O1 1".

If the variable after the "=" is a complex expression (e.g. A+Y), the expression will not be evaluated and the value of the first variable (e.g. A) will be used.

PMODE

Syntax

PMODE *graphics mode [, start page]*

Purpose

This statement sets the graphics mode for high resolution graphics. The following modes are available:

pmode	resolution	colors	pages used
0	128 x 96	2	1
1	128 x 96	4	2
2	128 x 192	2	2
3	128 x 192	4	4
4	256 x 192	2	4

Arguments

graphics mode may be a numeric expression which must evaluate to between 0 and 4.

start page may be a numeric expression which must evaluate to between 0 and 8. This value must represent a page that was previously reserved with a PCLEAR statement, and must also be greater than or equal to the number of reserved graphics pages, minus the number of pages required for the mode specified. If *start page* is omitted, the current start page (default = page 1) will be used.

Potential Errors

FC - *graphics mode* or *start page* are out of range; or an insufficient number of graphics pages have been reserved.

Examples

```
PMODE 4

PMODE 2, 5
```

POINT

Syntax

numeric variable = POINT (*X*, *Y*)

Purpose

This function returns the value of a specified pixel on the low resolution graphics screen (the normal text screen). If the pixel is part of a text character (ASCII values 0 to 127), a value of -1 will be returned; otherwise the color value of the pixel will be returned.

Arguments

X, which represents the horizontal component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 63.

Y, which represents the vertical component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 31.

Potential Errors

FC - *X* or *Y* is (are) out of range.

Examples

```
A = POINT (12, 3)

IF POINT (A, B*3) = -1 THEN PRINT "TEXT"

ON POINT (X, Y) GOTO 100, 200, 300
```

POKE

Syntax

POKE *memory location, value*

Purpose

This statement is used to define the contents of a specific memory location. Although used primarily for placing machine language program instructions into memory (to be used in conjunction with a BASIC program), it can also be used for graphics. On the normal text screen, POKE is the only statement that will allow certain characters (particularly inverse video characters) to be displayed.

Arguments

memory location may be a numeric expression which must evaluate to between &H0000 and &HFFFF (0 and 65535).

value may be a numeric expression which must evaluate to between 0 and 255.

If a rational number value is supplied for either argument, only the integer portion of the number will be used.

Potential Errors

FC - *memory location* or *value* is out of range.

Examples

```
POKE 16345, 23

POKE 1024, ASC("D")
```

Notes/Suggestions

The following short program displays all of the graphics characters available on the text screen using both POKE and CHR$. This serves to demonstrate the differences in screen output between the two methods.

```
00010 FOR C = 0 TO 255
00020 PRINT@C, CHR$ (C);
00030 NEXT C
00040 P = 256
00050 FOR C = 0 TO 255
00060 POKE P + C + 1024, C
00070 NEXT C
00080 GOTO 80
```

POS

Syntax

numeric variable = POS *(device number)*

Purpose

This function returns the current column position of the printer head or the cursor on the screen. The returned value can be examined to see if a line feed is required before another item is printed.

Arguments

device number may be an expression which must evaluate to one of the following values:

-2	- printer
-1	- cassette
0	- screen
1 to 15	- disk

Note, however, that the returned value is only significant when *device number* is 0 (screen) or -2 (printer). If any of the other allowable *device numbers* is specified, POS will always return a value of 0.

Potential Errors

DN - *device number* is not a legal file buffer.

NO - *device number* is a legal file buffer argument, but the file is not open.

Examples

```
PRINT POS (0)

A = POS (-2)
```

Notes/Suggestions

This short program will print text on the screen without word "wraparound".

```
00010 FOR T = 1 TO 18
00020 READ A$
00030 IF 32 - POS (0) < LEN (A$) THEN PRINT
00040 PRINT A$;
00050 IF POS(0) = 0 THEN PRINT " ";
00060 NEXT T
00070 DATA THIS, SHORT, TEST, WILL, SHOW, HOW
00080 DATA THE, POS, FUNCTION, CAN, BE, USED
00090 DATA TO, FORMAT, TEXT, ON, THE, SCREEN.
```

PPOINT

Syntax

numeric variable = PPOINT (*X, Y*)

Purpose

This function returns the current color of a pixel on the high resolution graphics screen. In the two color modes this will aways be a 0, 1 or 5 (depending upon the last color set selected). In four color modes, a number between 1 and 8 will be returned.

Arguments

X, which is the horizontal component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 65535, although BASIC will modify the value to between 0 and 255.

Y, which is the vertical component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 255, although BASIC will modify the value to between 0 and 191.

Potential Errors

FC - *X* or *Y* is out of range.

Examples

```
A = PPOINT (128, 96)

IF PPOINT (X, Y) = 4 THEN SOUND 100, 2
```

PRESET/PSET

Syntax

PSET *(X, Y [, color])*
PRESET *(X, Y)*

Purpose

This statement allows you to set (turn on) or reset (turn off) a specific pixel on the high resolution graphics screen. PSET allows a *color* to be specified, but if *color* is not specified, the current foreground color is used. PRESET resets a pixel to the current background color.

Arguments

X, which is the horizontal component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 65535, although BASIC will modify the value to between 0 and 255.

Y, which is the vertical component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 255, although BASIC will modify the value to between 0 and 191.

color may be a numeric expression which must evaluate to between 0 and 8.

Potential Errors

FC - *X*, *Y*, or *color* is out of range.

Examples

```
PSET (X, Y, 3)

PSET (126, 98)

PRESET (A+3, B*4)
```

PRINT

Syntax

PRINT *[# device,] [argument] [delimiter] [...]*

Purpose

This statement will write the *argument* to the *device* specified. In the absence of an output device, the screen is used by default. If a semicolon (;) is used as a *delimiter* to terminate the statement, a carriage return will not be executed. If a comma (,) is used, spaces will be printed up to the next comma field position. BASIC will allow the use of "?" as an abbreviation for the PRINT statement.

Arguments

device may be a numeric expression which must evaluate to one of the following values:

```
-2        - printer
-1        - cassette recorder
 0        - screen
 1 to 15  - disk files
```

All of these buffers except for 0 (screen) and -2 (printer) must be properly OPENed for output in order for communication to take place.

argument may be any valid numeric or string expression.

delimiter may be either a semicolon (;) or a comma (,).

The actual effect of the print delimiters varies from one output device to another--

	Comma (,)	Semicolon (;)
Screen	no carriage return; tabs to the next column position	no carriage return
Printer	no carriage return; tabs to next column position, stored at memory location 153	no carriage return

Cassette	inserts some extra spaces at the end of the record	no effect
Disk	no carriage return; does a TAB(15)	no carriage return

Of particular importance here is the fact that carriage returns are not inserted into sequential disk files. This means that if the following PRINT statement was executed:

```
PRINT #1, "HARRY"; 345
```

and the data was subsequently read back by the statement:

```
INPUT #1, A$, A
```

an IE or FD error would occur, and A$ would contain the value "HARRY345". For this reason it is recommended that you use a separate PRINT statement for each data item when writing to sequential files. If the program is written for cassette it will be easy to update later to disk.

Potential Errors

DF - the PRINT # statement caused the buffer to be written out to disk, at which time the disk was found to be full.

DN - the *device number* is out of range.

NO - the output device or buffer has not been OPENed.

VF - VERIFY is on; the buffer has been written out to disk, but DOS is unable to read the sector back.

WP - the disk is write-protected.

Examples

```
PRINT A$

PRINT 3*4, A$; R

PRINT #-1, A$
```

Notes/Suggestions

It is possible to include TAB and USING as part of the syntax for the "PRINT #" statement, as in the following examples:

```
PRINT #-2, TAB (20); A$;

PRINT #1, USING "#####.##"; A
```

See PRINT TAB and PRINT USING for more details.

PRINT @

Syntax

PRINT @ *screen location, [argument [delimiter]] [...]*

Purpose

This statement is for displaying text and graphics characters at a specific position on the text screen. BASIC allows the use of "?" as an abbreviation for PRINT.

Arguments

screen location may be a numeric expression which must evaluate to between 0 and 511.

argument may be any numeric or string expression.

delimiter may be either a semicolon (;) or a comma (,).

Potential Errors

FC - *screen location* is out of range.

Examples

```
PRINT @ 235, A$

PRINT @ N * 3, A$, N;
```

Notes/Suggestions

It is possible to include USING as part of the syntax for the "PRINT @" statement, as in the following example:

```
PRINT @224, USING "#####.##"; A
```

See PRINT USING for more details.

PRINT TAB

Syntax

PRINT TAB (*expression*) *[;]*

Purpose

This statement is used to position the screen cursor or the printer head prior to displaying data. You may use the abbreviated version of PRINT, which is "?".

Arguments

expression may evaluate to any number between 0 and 255.

Examples

```
PRINT TAB (8) A$

PRINT #-2, TAB (8) A; A$; TAB(24); B; B$
```

Potential Errors

FC - *expression* is out of range.

Notes/Suggestions

When TAB is encountered, the cursor is moved the number of positions specified by *expression* from the start of the original print line. This is best demonstrated by means of an example program:

```
00010 PRINT @ 200, "TEST";
00020 PRINT TAB (45);
00030 PRINT "TAB"
```

To determine where the word "TAB" will be printed, we must determine the print position of the first line in the print series. In this example, it is 192 (the first print position of the line containing screen position 200). Adding the value of *expression* (in this case, 45) to 192, we arrive at a "TAB" position of 237. The same logic is used when printing data on a printer.

Note that if the new position is less that the current cursor or printer head position, TAB will have no effect.

TAB can be included in the argument lists of PRINT # and PRINT @, as in the following examples:

```
PRINT #-2, TAB (A); A$

PRINT @224, A$; TAB (8); B$
```

See PRINT @ and PRINT # for more details.

PRINT USING

Syntax

PRINT USING *format string; agrument list [delimiter]*

Purpose

This statement prints text on the screen or printer in a predetermined format. It is particularly useful in printing columns of numeric data, and in formatting displays. BASIC allows the use of "?" as an abbreviation for the PRINT statement.

Arguments

format string may be any string literal or string variable containing valid format codes. The following codes are available:

FOR NUMERIC DATA --

#	specify the number of digits
.	include a decimal point
,	use commas to separate thousands, hundred thousands, etc.
**	fill leading blanks with asterisks
$	place dollar sign at beginning of field and spaces between "$" and first digit of number
$$	float dollar sign immediately in front of number
**$	fill blanks with asterisks and float dollar sign
↑↑↑↑	print in exponential format
+	display sign of number whether positive or negative
-	if "-" is leading character then always display it as a negative sign; if "-" is last character then display sign only when negative.

FOR STRING DATA --

!	display first character of string
%___%	select n characters from the beginning of the string, where "n" is equal to the number of blanks plus 2
chars	display as string literals

argument list may be a combination of either numeric expressions or string data, but each *argument* in the list must match *format string*. *Agruments* can be arranged in a list seperated by commas (,). If there are more arguments than fields in *format string*, then *format string* will be repeated.

Delimiter - can be a comma (,) or a semicolon (;) to end the *argument list*.

Potential Errors

% - not a bona-fide error, this indicates field overflow (the data is too large for the format string).

TM - the arguments and the format string do not match (ie. one is string data, the other numeric).

Examples

NUMERIC DATA --

In the following table, we have shown what happens if you issue a command like

```
PRINT USING A$; A
```

where A$ is equal to the format string in the left column, and A is set equal to either 1234.56 or -1234.56.

Format string	1234.56	-1234.56	
#####	1235	-1235	(note rounding)
#####.##	1234.56	-1234.56	
#####.####	1234.5600	-1234.5600	
####.##	1234.56	%-1234.56	(field overflow)
####,#.##	1,234.56	-1,234.56	
**#####.##	***1234.56	**-1234.56	
$#####.##	$ 1234.56	$-1234.56	
**$#####.##	***$1234.56	**$-1234.56	
##.##↑↑↑↑	1.23E+03	-1.23E+03	
#####↑↑↑↑	1235E+00	-1235E+00	(note rounding)
+####.##	+1234.56	-1234.56	
####.##+	1234.56+	1234.56-	
-#####.##	- 1234.56	--1234.56	(double "-")
####.##-	1234.56	1234.56-	

STRING DATA --

In the following examples, A$ = "Bob", B$ = "Dave" and a "_" represents a blank.

```
PRINT USING "%_%"; A$, B$

     BobDav

PRINT USING "%__%_and_%__%"; A$, B$

     Bob__and_Dave

PRINT USING "!'s_Score_=_####" + CHR$(13) +
     "!'s_Score_=_####";A$, 123, B$, 456

     B's_Score_=__123
     D's_Score_=__456
```

Notes/Suggestions

PRINT USING may be used in conjunction with PRINT @ or with PRINT #, as in the following examples:

```
PRINT @224, USING "#####.##"; A

PRINT #-2, USING "#####.##"; A
```

See PRINT @ and PRINT # for more details.

PUT

Syntax

PUT *(X1, Y1) - (X2, Y2), variable [, action]*

Purpose

This statement is used in conjunction with the GET statement to move graphics on the high resolution graphics screen. When properly used, fairly fast animation is possible.

Arguments

X1 and *X2* may be numeric expressions which must evaluate to between 0 and 65535, although BASIC will modify the values to between 0 and 255.

Y1 and *Y2* may be numeric expressions which must evaluate to between 0 and 255, although BASIC will modify the values to between 0 and 191.

variable must be a numeric array variable that has been previously dimensioned.

action - The follow options are allowable as actions, as long as the "G" option was used with the GET statement:

PSET - sets the pixels that were set in the original rectangle.

PRESET - resets the pixels that were set in the original rectangle.

AND - compares each pixel of the orginal rectangle with that in the destination rectangle. If both pixels are set, then the pixel in the destination rectangle will be set; if not, the pixel will be reset.

OR - compares each pixel of the original rectangle with that in the destination rectangle. If either pixel is set, the pixel in the destination rectangle will be set; otherwise it will be reset.

NOT - reverses (complements) the state of each pixel in the destination rectangle regardless of the contents of the PUT array.

Potential Errors

FC - the size of the rectangle is larger than the array size; *X1*, *X2*, *Y1*, or *Y2* is out of range; or *variable* has not been dimensioned.

Examples

```
PUT (A, B) - (A+30, B+30), GR, PRESET

PUT (20, 20) - (30, 30), DD
```

Notes/Suggestions

If garbage is appears on the screen, instead of the data from the GET statement, one of the following conditions may be present:

1. The "G" option was used with GET and *action* was not used with PUT.
2. The "G" option was not used with GET and *action* was used with PUT.
3. The size of the GET and PUT rectangles do not match.
4. Part of the PUT rectangle is off the edge of the screen; eg. PUT (180, 180) - (200, 200), A

PUT

Syntax

PUT *#buffer [, record number]*

Purpose

This statement is used with direct (random) access disk files to transfer data from a specified open *buffer* onto the disk itself. PUT # is employed after data has been placed in the *buffer* by means of either the PRINT, WRITE, LSET or RSET statement.

Arguments

buffer may be a numeric expression which must evaluate to an OPEN disk file buffer in the range 1 to 15.

record number may be a numeric expression which must evaluate to between 1 and 32767. If no record number is specified, the current record (see LOC) is used. LOC is incremented after every PUT.

Potential Errors

BR - *record number* is 0, or *record number* has caused the file size to exceed 612 physical disk sectors.

DF - the PUT statement caused the buffer to be written out to disk, at which time the disk was found to be full.

DN - the *buffer* specified refers to a disk buffer not reserved with FILES, or *buffer* is out of range (but less than 32768).

FC - *record number* is less than 0 or greater than 32767; or *buffer* is less than 1 or greater than 32767.

NO - the *buffer* specified has not been OPENed.

VF - (VERIFY is on) you have PUT a record which caused a disk write to take place and the subsequent read attempt failed.

WP - you have attempted to PUT a record on a disk which is write-protected.

Examples

```
PUT #1

PUT #D, R

PUT #8, LOF(8) + 1
```

Notes/Suggestions

If *record* is greater than the last record in the file (e.g. "PUT#1, 100" when the last record is currently #50), the file will be enlarged to accommodate the specified record. The records in between (in this case, records 51 to 99) will contain the information that was on the disk at the time of the PUT -- in other words, garbage. If you are going to write data in a non-sequential manner, be sure to set all the records to some type of "null" value when opening the file for the first time.

READ

Syntax

READ *variable [, variable] [...]*

Purpose

This statement reads numeric or string values in sequential order from corresponding DATA statements and assigns the values to the specified *variables*.

Arguments

variable may be any string or numeric variable.

Potential Errors

OD - there is no more data to READ

SN - an attempt is being made to assign string data to a numeric variable. The line number in the error message is the line number of the DATA statement containing the string data.

Examples

```
READ A$

READ A

READ A, A$, B, B$
```

RELATIONAL OPERATORS

Syntax

1. Equal To
[numeric variable =] expression = expression

2. Less Than
[numeric variable =] expression < expression

3. Greater Than
[numeric variable =] expression > expression

4. Less Than Or Equal To
[numeric variable =] expression <= expression

or

[numeric variable =] expression =< expression

5. Greater Than Or Equal To
[numeric variable =] expression >= expression

or

[numeric variable =] expression => expression

6. Not Equal To
[numeric variable =] expression <> expression

or

[numeric variable =] expression >< expression

Purpose

These relational operators are used whenever you want to compare two *expressions* to determine if they are equal or whether one is larger than the other. The result of the comparison is converted to a Boolean value of 0 for false or -1 for true. Although such operators are normally used in conjunction with the IF-THEN-ELSE statement, the result can easily be passed to a *numeric variable* and used for any other purpose. (See *IF-THEN-ELSE* for more details.)

Arguments

expression may be either a string expression or a numeric expression, but both *expressions* must be of the same type. If numeric, *expression* will be limited according to which terms (COS, LOG, SQR, etc.) it contains, but under most circumstances, it may take on any positive or negative value in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
A = A = B

A = A$ = B$

IF SIN (A) < 0 THEN ....
```

Notes/Suggestions

Whenever possible, avoid the use of "<=", ">=", and "<>", since each of these notations requires extra memory for storage, and BASIC spends a little more time evaluating the relation. There is almost always a way to rewrite the comparative test so that less time and memory are used up. For example, suppose you were testing for a non-zero value:

```
00010 IF A <> 0 THEN 30
00020 PRINT "FALSE"
00030 ...program continues...
```

If the test is successful, control is transferred to line 30. Why not save 7 bytes of memory and marginally increase the speed of the program by testing for 0 instead, and rearranging the program a little:

```
00010 IF A = 0 THEN PRINT "FALSE"
00020 ...program continues...
```

RENAME

Syntax

RENAME *current name* TO *new name*

Purpose

This statement is used to change the name of a disk file or program as it physically appears on the disk.

Arguments

Both *current name* and *new name* must be legal disk *file specifications*. A *file specification* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (","), slash ("/"), or colon (":").

extension - any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). For the RENAME statement, the *extension* must be supplied.

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive. Note that the drive number must be the same for both names.

Potential Errors

AE - *new name* already exists on the disk.

FC - the drive numbers in *current name* and *new name* do not match.

FN - one of the *names* is illegal.

IO - the disk is not properly inserted in the drive; the drive door is not closed; the controller is unable to read the directory track.

NE - *current name* does not exist on the disk in the drive specified.

VF - VERIFY is on; after writing the sector containing *new name* back to disk, the controller was unable to read back that sector.

WP - the disk is "write protected".

Examples

```
RENAME "PROGRAM.BAS" TO "GAME.BIN"

RENAME "OLD.DAT:1" TO "NEW.DAT:1"

RENAME F$(8) TO F$(9)
```

RENUM

Syntax

RENUM *[newline] [, startline] [, increment]*

Purpose

This statement is used to renumber some or all of the lines in a BASIC program. As well, it correctly renumbers line numbers referenced by GOTO, GOSUB, THEN, ON...GOTO and ON...GOSUB statements. Note that RENUM cannot be used to rearrange the order of lines, only to change their numbers.

Arguments

newline is the new line number of the first line in the program to be renumbered, and can be any valid line number from 0 to 63999. If omitted, number 10 is used by default.

startline is the first line to be renumbered, and can be any valid line number from 0 to 63999. If omitted, the first line number in the program is used.

increment is the rate at which line numbers are to increase in size, and can be any integer value from 1 to 63999, as long as the *increment* does not cause any line number to be above 63999. If omitted, an increment of 10 is used by default.

Potential Errors

FC - an attempt has been made to change the order of lines; the *increment* is out of range; or the *increment* has caused the appearance of a line number greater than 63999. (If this error occurs, the line numbers in the program will not be altered.)

SN - a line number greater than 63999 has been encountered following a GOTO, GOSUB etc. (If this error occurs, you may find that your program has been converted into garbage!)

UL - a line referenced in a GOTO, GOSUB, etc. did not exist in the program before renumbering. (If this error occurs, the program will still be completely renumbered and the GOTOs, etc. will now reference non-existent or wrong lines.)

Examples

```
RENUM

RENUM 1000, 400, 10

RENUM , , 1

RENUM 1000, , 1
```

Notes/Suggestions

To avoid the frustration of discovering a ream of UL errors during a RENUM, use the following statement before actually attempting to renumber the program:

```
RENUM 63999, 63999
```

BASIC will go through the entire program and report any UL errors; however, since both *startline* and *newline* are greater than the highest line in the program (or the same if you happened to have line 63999 in your program) the program in memory will be unchanged. You can now go through the progam and correct the affected lines, before the line numbers are actually changed. If you wish to have the list of undefined line numbers sent to the printer, use the following:

```
POKE &H6F, &HFE: RENUM 63999, 63999.
```

RESTORE

Syntax

RESTORE

Purpose

This statement is used to set BASIC's DATA pointer back to the beginning of the program. Following a RESTORE, the next item to be read by a READ statement will be the first DATA item in the program.

Arguments

None.

Potential Errors

None.

Examples

```
RESTORE

IF A$ = "XXX" THEN RESTORE
```

RETURN

SEE *GOSUB-RETURN*

RIGHT$

Syntax

string variable = RIGHT$ *(string expression, position)*

Purpose

This function is used to obtain the rightmost portion of *string expression*, from a given starting *position*.

Arguments

string expression may be any combination of from 0 to 255 characters, each of which may have any ASCII value from 0 to 255.

position may be a numeric expression which must evaluate to between 0 and 255. If *position* is greater than the length of *string expression*, the entire *string expression* will be returned.

Potential Errors

FC - *position* is out of range.

Examples

```
A$ = RIGHT$ (A$ + CHR$ (13) + B$, X)

IF RIGHT$ (B$, 3) = "XXX" THEN ...
```

RESET

SEE *SET/RESET*

RND

Syntax

numeric variable = RND *(expression)*

Purpose

This function returns a positive pseudo-random number, the magnitude of which depends on the nature of *expression*, and also causes the random "seed" to be updated for subsequent calls. It is called a "pseudo-random number" because, although the sequence of numbers generated exhibits all the qualities of a list of true random numbers, the list can be predicted whenever the random "seed" is known.

Arguments

expression may evaluate to any positive or negative number in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
PRINT RND (8)

A = RND (0) * 10

R = RND (-X)
```

Notes/Suggestions

If *expression* is a positive number greater than or equal to 1, then the random number will will always be a positive integer number in the range 1 to *expression*.

If *expression* is a positive number less than 1, then the random number will always be a positive rational number between 0 and 1.

If *expression* is any negative number, then the random number will always be a positive rational number between 0 and 1, but note that for every potential negative value of *expression*, there is precisely one value that will be returned. For example, RND (-1) will always return the same value of .522222849.

When the computer is first turned on, the random "seed" is always set to the same value, and for this reason, RND will return the same series of values in the same order. To overcome this problem (which causes random-generated games to always run the same way), use a command like the following in your program:

```
00010 T = RND (-TIMER)
```

This causes the random "seed" to be more randomly reset each time the program is run. The "-" sign is used to force the computer to select a rational number instead of an integer number.

RSET

Syntax

RSET *fielded string variable = string expression*

Purpose

This statement is used in conjunction with direct (random) access disk files. It transfers and right justifies *string expression* into *fielded string variable*. If the length of *string expression* is greater than the field size for the specified variable, then the excess characters are truncated. If the length of *string expression* is less than the field size for the specified variable, then the new string is blank-filled on the left. The primary purpose of this statement is to prevent field overflow when writing new records to the file.

Arguments

string expression must evaluate to any combination of from 0 to 255 ASCII characters, each of which may have any ASCII value from 0 to 255.

fielded string variable may be any legitimate string variable name which has been previously defined in a *FIELD* statement.

Potential Errors

SE - you have named a string variable which has not been previously fielded.

Examples

```
RSET A$ = B$

RSET A$ = "NOW IS THE TIME ..."

RSET A$ = MKN$ (A)
```

Notes/Suggestions

See *FIELD* for more information concerning the use of fielded variables.

RUN

Syntax

RUN *[line number]*
RUN *[filespec]*

Purpose

This statement is used to start the execution of a BASIC program. If the optional *line number* is included, the program will start execution at the line specified. If you have a disk system, you may also supply the *filespec* option, which causes the specified program to be loaded from the disk and to be executed starting at the first line. Note that if the former (RUN *[line number]*) type of command is used, all OPEN files are left OPEN; otherwise, files are CLOSEd before execution takes place.

Arguments

line number may be any valid BASIC line number from 0 to 63999.

A *filespec* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (","), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extension "BAS".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

Potential Errors

FM - you have made an attempt to load a file which is not a BASIC program.

FN - the specified *filespec* is illegal.

IO - the disk is not properly inserted in the drive; the drive door is not closed; or the data on the disk is unreadable.

NE - *filespec* does not exist on the disk.

UL - the *line number* specified does not exist.

Examples

```
RUN

RUN 100

RUN "GAME"

RUN "GAME.NEW:1"
```

Notes/Suggestions

If you are using the RUN statement within one BASIC program to call another program, you must use a string literal for *filespec*. If you need to use a variable name, then use the LOAD statement with the ",R" option instead. LOAD A$, R and LOAD "PROGRAM" , R and RUN "PROGRAM" all work; RUN A$ does not.

SAVE

Syntax

SAVE *filespec*

Purpose

This statement causes the BASIC program currently in memory to be written out to disk. If the specified *filespec* already exists on the disk, the new file will over-write the old one. Note that OPEN files are left OPEN.

Arguments

A *filespec* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (".), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extension "BAS".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

Potential Errors

DF - the disk you are saving the program on is full. If you get this error, check the disk directory. If any room was available when the SAVE started, some of the program will be SAVEd and a directory entry will be made; however, you will not be able to LOAD any of the program.

FN - *filespec* is not a legal disk file specification.

IO - the disk is not properly inserted in the drive; the drive door is not closed; when scanning the directory for a *filespec* match, the controller was unable to read a sector; or, a write error occurred.

VF - VERIFY is on; following a sector write; the controller was unable to successfully read back the same sector.

WP - the disk you are SAVEing to has a "write protect" tab on it.

Examples

```
SAVE "GAME"

SAVE "GAME.BBB"
```

SAVEM

Syntax

SAVEM *filespec, start, end, entry*

Purpose

This statement causes a binary machine-language file to be written out to disk. It duplicates the contents of memory from *start* to *end* inclusive. *Entry* must be provided so that when the file is re-loaded, BASIC will be able to EXECute the program. If *filespec* already exists on the specified disk, the old file will be over-written by the new file. Note that OPEN files are left OPEN.

Arguments

A *filespec* consists of 3 parts:

filename - any character string of from 1 to 8 characters in length; the ASCII value of each character must be between 1 and 255 except that the value may not correspond to period (","), slash ("/"), or colon (":").

extension - (optional) any character string of from 1 to 3 characters in length; the same ASCII requirements as for the filename must be met. The extension is separated from the filename by a period (".") or a slash ("/"). If no extension is included, BASIC will use the default extension "BIN".

drive - (optional) must be a number from 0 to 3. The drive number may appear at the start or the end of the filespec; it is always separated from the body of the filespec by a colon (":"). If no drive number is specified, BASIC will use the current default drive.

start, *end*, and *entry* must be valid addresses in the range &H0000 to &HFFFF (0 to 65535).

Potential Errors

DF - the disk you are saving the program on is full. If you get this error, check the disk directory. If any room was available when the SAVEM started, some of the program will be saved and a directory entry will be made. You will be able to LOADM the portion of the file that was saved, but you will get an "?IE ERROR".

FC - *start*, *end*, or *entry* values are out of range.

FN - *filespec* is not a legal disk file specification.

IO - the disk is not properly inserted in the drive; the drive door is not closed; when scanning the directory for a *filespec* match, the controller was unable to read a sector; or, a write error occurred.

VF - VERIFY is on; following a sector write; the controller was unable to successfully read back the same sector.

WP - the disk you are SAVEing to has a "write protect" tab on it.

Examples

```
SAVEM "GAME", 12345, 23456, 12345

SAVEM "GAME.BBB", &H0E00, &H0F00, &H0F12
```

Notes/Suggestions

SAVEM can be used to save a graphics screen to disk, to be loaded later by a BASIC program. The following example will save the current graphics screen (notice how we prevent inadvertant execution of this file by specifying an *entry* point of &HA027, which is BASIC's RESET entry address):

```
50000 P = PEEK (&HBA) * 256 + PEEK (&HBB)
50010 M = PEEK (&HB6)
50020 IF M = 0 THEN M = 1536
      :ELSE IF M = 1 OR M = 2 THEN M = 3072
      :ELSE M = 6144
50030 SAVEM "SCREEN", P, P + M, &HA027
```

To re-load the screen just saved, first make sure the computer has been set up for the same graphic configuration that existed when the SAVEM was done. Then use the following to load the data:

```
50050 SCREEN 1, 1
50060 LOADM "SCREEN"
```

SCREEN

Syntax

SCREEN *mode [, colorset]*

Purpose

This statement is used to select either the graphic screen or the text screen as well as one of the two different *colorsets* available.

Arguments

mode may be a numeric expression which must evaluate to between 0 and 255. If *mode* is 0, then the text screen will be selected. Any other value will cause the graphics screen to be selected.

colorset may be a numeric expression which must evaluate to between 0 and 255. If it is omitted, then BASIC will use one of two defaults, depending on the value of *mode*. If *mode* is 0 then the default *colorset* is also 0; if *mode* is 1, then BASIC will use the last defined *colorset*.

Potential Errors

FC - either *mode* or *colorset* is out of range.

Examples

```
SCREEN 1, 0

SCREEN 1

SCREEN 0, 1
```

Notes/Suggestions

Normally, BASIC automatically returns to SCREEN 0, 0 (the normal text screen) when a PRINT or INPUT statement is encountered, when an error occurs, or when the end of the program is reached. You can disable this automatic return by means of the statement "POKE 359, 57". To return to normal operation, use the statement "POKE 359, 126".

You can obtain a few additional graphic modes not normally accessible through BASIC by POKEing values into memory location &HC1, which is where the SCREEN *colorset* values are normally stored. The value to POKE must be between 0 and 255 and should be a multiple of 8 for best results. Once the new value has been POKEd, simply execute a "SCREEN 1" command to cause the graphic display to change. You will discover, as you experiment with this technique, that the results will vary depending on which PMODE has been selected.

SET/RESET

Syntax

SET *(X, Y, color)*

RESET *(X, Y)*

Purpose

These statements are used to control individual pixels on the low resolution (text) screen. SET causes the specified pixel to be illuminated with the specified *color*. RESET will turn the specified pixel off, and is, in effect, a SET with a *color* value of 0. These commands produce the best results when the screen has been cleared to black by means of a "CLS 0" command.

Arguments

X, which is the horizontal component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 63.

Y, which is the vertical component of a co-ordinate pair, may be a numeric expression which must evaluate to between 0 and 31.

color may be a numeric expression which must evaluate to between 0 and 8.

Potential Errors

FC - *X*, *Y*, or *color* is out of range.

Examples

```
SET (1, 2, 3)

SET (X, Y, C+1)

RESET (X * 3, Y / 3)
```

Notes/Suggestions

Each PRINT @ screen location contains 4 pixels which can be SET or RESET. All SET pixels in the same screen location must be SET to the same color. This is a hardware limitation and has nothing to do with software. For example, if you SET the pixel at (30, 40) to color #2 and subsequently SET the pixel at (31, 40) to color #4, both pixels would end up being SET to color #4 because they are both in the same PRINT @ screen location.

SGN

Syntax

numeric variable = SGN *(expression)*

Purpose

This function is used to determine the sign of a number. It returns a value of -1 if *expression* is negative, 0 if *expression* is equal to 0, and 1 if *expression* is positive. SGN may be used in conjunction with ABS to separate any number into its sign and magnitude parts.

Arguments

expression may be any positive or negative value in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
A = SGN (A)

ON SGN (A) + 2 GOTO 100, 200, 300
```

SIN

Syntax

numeric variable = SIN *(expression)*

Purpose

This function returns the sine of *expression*, which is assumed to be a radian angle. The returned value is always a rational number between -1 and +1.

Arguments

expression may be any positive or negative value in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
A = SIN ((90 * (2 * PI)) / 360)

IF SIN (X) < 0 THEN ...
```

Notes/Suggestions

See *COS* for an explanation of the conversion of angles from degrees to radians and vice versa.

SKIPF

Syntax

SKIPF *[filename]*

Purpose

This statement allows you to search a cassette tape for the end of a file called *filename*. If *filename* is omitted, the cassette will stop at the end of the first file encountered. When a file is found, the name is displayed at the top of screen, along with the flashing "S". If this name is the same as the specified *filename*, the "S" turns into an "F".

Arguments

filename may be any character string of from 0 to 8 characters in length. Each character may have any ASCII value from 0 to 255. If the *filename* exceeds 8 characters in length, it will be truncated to the 8 character limit.

Potential Errors

IO - the data on the tape is unreadable.

Examples

```
SKIPF

SKIPF "GAME"

SKIPF F$
```

Notes/Suggestions

The SKIPF statement provides a convenient means of verifying that a tape file has been correctly written, and is particularly useful following a CSAVE or CSAVEM statement. After the program has been saved, rewind the tape and use the SKIPF instruction to read over the file. If no "?IO ERROR" occurs, you will know that the program has been correctly saved; otherwise, you can easily repeat the CSAVE or CSAVEM instruction, since the program will still be intact in memory.

To see what programs are stored on an unlabeled tape, use the SKIPF instruction as follows:

```
SKIPF "XXXXXX"
```

The computer will scan the entire tape, displaying all files as they are encountered. The tape player will not stop unless the file called "XXXXXX" happens to be on the tape or an "?IO ERROR" occurs, and therefore, once the end of the tape is reached, you will have to press the RESET button.

SOUND

Syntax

SOUND *pitch, duration*

Purpose

This statement causes a tone of specifed *pitch* and *duration* to be played through the monitor's speaker.

Arguments

pitch may be a numeric expression which must evaluate to between 0 and 255. The higher the value, the higher the tone.

duration may be a numeric exression which must evaluate to between 0 and 255. The greater the value, the longer the tone will be sounded.

Potential Errors

FC - *pitch* or *duration* is out of range.

Examples

```
SOUND A / 3, B * 2

SOUND 255, 255
```

SQR

Syntax

numeric variable = SQR *(expression)*

Purpose

This function is used to determine the square root of a given *expression*.

Arguments

expression may be positive value in the range 10^{-38} to 10^{+38}.

Potential Errors

FC - *expression* is a negative number.

Examples

```
PRINT SQR (16)

A = SQR (1234)
```

Notes/Suggestions

Even though the exponentiation (↑) function can be used to find a square root (A = 16↑(1/2) is the same as A = SQR (16)), we recommend that you use the SQR function whenever possible, since it takes less time to execute. However, if you need to determine a cube root (or any other root) you must use the exponentiation function. For example,

```
PRINT 125↑(1/3)
```

will print "5", the cube root of 125.

STOP

Syntax

STOP

Purpose

This statement acts as a "breakpoint" and causes the execution of a BASIC program to be temporarily halted. You can use it during the debugging phase of program development by including it within a program line; once the "BREAK" message appears, you can examine the variables to ensure that they all hold the correct values. Then, when you are satisfied, you can resume execution by means of the CONT statement. Note that all OPEN files are left OPEN.

Arguments

None.

Potential Errors

None.

Examples

```
100 STOP
```

STR$

Syntax

string variable = STR$ *(expression)*

Purpose

This function is used to convert a numeric value into its string equivalent. The logical inverse of this function is VAL, which converts a string into its numeric equivalent.

Arguments

expression may be any positive or negative value in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
A$ = STR$ (A)
```

Notes/Suggestions

When *expression* is a positive number, the STR$ function adds a leading blank to the resulting string. The following program shows how this blank can be removed.

```
00010 A$ = STR$ (A)
00020 IF LEFT$ (A$, 1) = CHR$ (32) THEN A$ = MID$ (A$, 2)
```

STRING$

Syntax

string variable = STRING$ *(expression, character)*

Purpose

This function defines a character string of length *expression*, in which all characters are identical.

Arguments

expression must evaluate to any number between 0 and 255.

character may be a string variable, a string enclosed in quotes, or an ASCII code value between 0 and 255.

If *character* is a string that exceeds 1 character in length, then only the first character will be used.

Potential Errors

FC - *character* is a numeric expression outside allowable ASCII limits.

OS - not enough string space has been cleared.

Examples

```
PRINT STRING$ (5, "*")

PRINT STRING$ (5, 42)

A$ = CHR$(42): PRINT STRING$(5, A$)
```

Notes/Suggestions

Color BASIC does provide a reverse tab capability; however, you can artificially produce a reverse tab by using STRING$. For example,

```
PRINT STRING$ (9, 8);
```

will backspace the cursor nine spaces (note that the ";" is required to prevent a carriage return). This technique can be used any time it is necessary to reposition the cursor (as might be required following an incorrect data entry):

```
00010  INPUT"PLAY AGAIN y/n"; I$
00020  IF I$ <> "Y" AND I$ <> "N" THEN PRINT STRING$
       (32, 8);
      :GOTO 10
00030  ' PROGRAM CONTINUES
```

TAB

SEE *PRINT TAB*

TAN

Syntax

numeric variable = TAN *(expression)*

Purpose

This function returns the tangent of *expression*, which is assumed to be a radian angle.

Arguments

expression may be any positive or negative value in the range 10^{-38} to 10^{+38}.

Potential Errors

None.

Examples

```
A = TAN (RD)

IF TAN (A) < 0 THEN ...
```

Notes/Suggestions

The TAN function is quite accurate in all quadrants except when the angle approaches to within 2 or 3 degrees of 90 or 270 degrees (on either side of the asymptote). The true tangent of such angles begins to rapidly climb towards infinity, and of course BASIC is not able to handle infinitely large numbers. To prevent overflow from occurring, BASIC has imposed a limit on the output that results in a maximum tangent value of approximately 13,500,000 (positive or negative).

TIMER

Syntax

numeric variable = TIMER
TIMER = *expression*

Purpose

This dual purpose function makes use of the 60 Hertz interrupt capability of the 6809 processor. Each time an interrupt occurs (at a processor speed of 1 megaHertz, it occurs 60 times per second), an interrupt service routine is entered, in which a two-byte counter is incremented. This counter value is passed to *numeric variable* when TIMER appears on the right side of the equals sign. When TIMER appears on the left side of the equals sign, the counter is given a new value equal to *expression*. Whenever cassette, disk, and RS-232 I/O is in progress, the interrupt is disabled, so TIMER is only accurate during cassette, disk and RS-232 inactivity.

Arguments

expression may be any positive value in the range 0 to 65535.

Potential Errors

FC - *expression* is out of range.

Examples

```
TIMER = 0

TM = TIMER

SC = TIMER / 60

MN = TIMER / 60 / 60
```

Notes/Suggestions

Examples #2 and #3 above show how the timer value can be converted to seconds and minutes respectively. If the processor did run at exactly 1 megahertz, these two examples would be exactly correct. Unfortunately, in the Color Computer, the 6809 runs at a slightly slower speed, typically between 0.89 MHz and 0.93 MHz. This means that the conversion factor will necessarily be different from 60. We have found that the interrupt routine is actually entered between 55 and 56 times per second on our machines and so, for us, a more accurate conversion factor is about 55.5. You will have to experiment for yourself to determine the correct conversion factor for your machine.

TRON/TROFF

Syntax

TRON
TROFF

Purpose

These statements are used to enable (TRON) and disable (TROFF) BASIC's line-trace function. When the trace is turned on, and a BASIC program is executed, the line numbers of the program are displayed on the text screen as each one is executed. This feature is useful during program debugging.

Arguments

None.

Potential Errors

None.

Examples

```
TRON

TROFF
```

UNLOAD

Syntax

UNLOAD [*drive*]

Purpose

This statement closes all data files on the *drive* specified. If *drive* is not specified, the files on the default drive (set by the DRIVE statement) are closed.

Arguments

drive may be a numeric expression which must evaluate to between 0 and 3.

Potential Errors

FC - *drive* is out of range.

Examples

```
UNLOAD 1

UNLOAD
```

Notes/Suggestions

WARNING: This function does not work properly with Disk BASIC Version 1.0. If more than one random access file is open, an UNLOAD will cause the computer to completely lock-up. This bug has been fixed in versions 1.1 and later, but because of this major bug, we do not recommend the use of this statement--use CLOSE instead.

USING

SEE *PRINT USING*

USR

Syntax

variable = USR *[number] (expression)*

Purpose

This function allows you to pass program control to a machine language subroutine whose entry address has been previously defined by the DEF USR statement. The subroutine will accept a single 16-bit (0 to 65535) parameter value defined by *expression*. If *number* is omitted, BASIC assumes that the USR0 function is to be used.

Arguments

variable may be either a numeric variable or a string variable, but it must be type compatible with *expression*.

number must be a digit from 0 to 9.

expression may be either a string expression (a character string of from 0 to 255 characters in length, each of which may have any ASCII value between 0 and 255) or a numeric expression which must evaluate to between 0 and 32767 (if a value between 32768 and 65535 is to be sent, it must be passed in the form "*expression* - 65536" or BASIC will not accept it.)

Potential Errors

FC - the numeric *expression* is out of range; or, the USR function has not been previously defined by a DEF USR statement.

Examples

```
A = USR0 (0)

A = USR1 (40000 - 65536)

A = USR2 (VARPTR (A$ (0)))

A$ = USR3 (STRING$ (255, " "))
```

Notes/Suggestions

The USR function is one of the most powerful commands available in Color BASIC. Because parameters may be passed to and from the USR routine, it can be used for almost any purpose where pure machine-language is to be preferred over interpreted BASIC.

As mentioned earlier, the entry address for the USR function is defined by means of the DEF USR statement. If you do not have Extended Color BASIC, however, then you do not have the luxury of this statement. Instead, you must POKE the two bytes of the entry address into the two memory locations starting at &H0113 (275 decimal)--the most significant byte of the address is stored at location 275 and the least significant byte of the address is stored at location 276. (In non-Extended systems, location 274 contains a "JMP" opcode byte, &H7E or 126.)

Suppose, for example, that you had entered a machine language routine into memory starting at address AD, and this address just happened to be the normal entry point. The following segment of BASIC code would properly set up your USR definition:

```
00100  MS = INT (AD / 256)          * get MS byte of address
00110  LS = INT (AD - MS * 256)     * get LS byte of address
00120  POKE 275, MS                 * define USR entry point
      :POKE 276, LS

... program continues ...
```

Once the entry point has been defined, you can safely make USR function calls from anywhere within your program.

Since only one parameter may be passed to the routine, generally the parameter will be a 2-byte address that points to the start of a string, or to a block of other pertinent data.

Once the USR call has been made, it is a simple matter for the subroutine to get the parameter by means of a "JSR $B3ED" call. This causes the parameter to be stored in the A and B accumulators (the D register), and the routine can do whatever is required with the data.

When the subroutine is complete, and it is time to return to the BASIC program from which it was called, return can be accomplished in one of two ways. The simplest way to return is via an "RTS" instruction, in which case no parameter is returned to the calling program. If it is necessary to return a parameter

to the calling program, you can do so by first loading the A and B accumulators (the D register) with the desired value and exiting via a "JMP $B4F4" (not a "JSR") instruction. In examples #1 to #3 above, the parameter will be returned to the variable on the left side of the equals sign (variable A).

As pointed out earlier, BASIC will not accept a parameter value greater than 32767. For this reason, it is necessary to "cheat" a little, or rather to "fool" BASIC into thinking the parameter is valid. This is accomplished by using the form "*expression* - 65536". As it turns out, if this format is used, the parameter value is not evaluated until the "JSR $B3ED" call is made, at which time it is still possible to get an "?FC ERROR" if the parameter is outside the range 0 to 65535.

In example #4 above, you can see that string data may be supplied as a "pseudo-parameter" for the USR call, as long as the variable name on the left of the equals sign is a string variable. This form of the USR call is quite peculiar in that you do not have to make an explicit call (JSR $B3ED) to obtain the parameter. Instead, on entry to your machine language routine, the X register is already initialized with the address of the descriptor of the string that was passed.

Although a little slower than the method described in the preceding paragraph, it is possible to pass the VARiable PoinTeR of a string variable to the USR function. The USR routine may then alter the contents of the string descriptor or the string itself. Extreme caution should be exercised when using this technique, however, because you may inadvertantly destroy part of the BASIC program. If you are certain that the string is located in the string pool and not within the body of the program, then modifications are relatively safe as long as you do not attempt to make a string longer than it was before the call was made.

If your BASIC program makes calls to one or more subroutines that do not accept or return parameters, then you should use EXEC instead of USR. EXEC is both faster and more space efficient than USR and, obviously, does not tie up any of the USR functions. Additionally, EXEC does not require that the routine(s) be previously defined by the "DEF USR" statement.

See the section of the Book entitled Machine-Language Subroutines for numerous examples of the USR function.

VAL

Syntax

numeric variable = VAL *(string expression)*

Purpose

This function is used to convert a string into its numeric equivalent. If the first character of *string expression* is not a number, a space, "+", "-", "&H" or "&O", VAL will return a value of 0.

Arguments

string expression may be any string literal or string variable.

Potential Errors

None.

Examples

```
PRINT VAL ("123")

PRINT VAL ("ABC123")

A = VAL (A$)
```

Notes/Suggestions

The following program shows how VAL can be used to convert hexadecimal numbers stored in a DATA statement to decimal numbers.

```
00010 DATA 12, AB, CD, FF
00020 FOR T = 1 TO 4
00030 READ A$
00040 A = VAL ("&H" + A$ )
00050 PRINT A
00060 NEXT T
```

VARPTR

Syntax

numeric variable = VARPTR *(variable name)*

Purpose

This function returns a 2-byte address which represents a pointer to the 5-byte block of memory assigned to *variable name*. The actual contents of the memory block will vary according to the type and nature of the variable. (See the notes below for more details.)

Arguments

variable name may refer to any legitimate variable, including string and numeric variables, simple or array type.

Potential Errors

None.

Examples

```
A = VARPTR (B$)

A = VARPTR (B$ (0, 0))

A$ = HEX$ (VARPTR (B$))
```

Notes/Suggestions

The value that is returned following a VARPTR command always points to a 5-byte block of memory that is occupied by the specified variable. If the variable is numeric (whether it is simple or array), then the block contains the the actual value of the variable, in floating point format, where byte #0 represents the exponent and bytes #1 through #4 represent the mantissa. On the other hand, if the variable was a string variable (again, it can be either simple or array), then the block actually represents a descriptor for the string. In this case, byte #0 of the block specifies the length of the string, and bytes #2 and #3 contain the address of the first character in the string. In this descriptor, bytes #1 and #4 are not used.

Every variable also includes a short header block that contains the variable name and other pertinent information. In the case of simple variables, the header is always 2 bytes long (it contains only the variable name) and immediately precedes the 5-byte data block in memory. In the case of array variables, however, the header block is quite different--its length will always be 5 bytes plus 2 bytes for each dimension--and it is located immediately preceding the 5-byte data block for the lowest numbered element in the array. (See Program Optimization Techniques for more details on variable storage.)

VARPTR is particularly useful in passing a variable address to a USR function. The subroutine can then access any portion of the variable and modify the data if required.

One thing to be aware of when using this function is that the introduction of a new simple variable causes the array variables to be moved upward in memory. This can have serious ramifications when a VARPTR statement is used to get the address of an array variable. Consider the following example:

```
A = VARPTR (A$ (0))
```

This is a perfectly acceptable statement and will work correctly as long as A has been previously defined. If A has not been defined, however, the value returned will be 7 bytes lower than the actual location of A$ (0). This is because the address was calculated first, and then the array variable had to be moved up in memory to make room for the simple variable. Note that this problem does not occur if you are getting the pointer for a simple variable. Obviously the solution is to ensure that all variables are dimensioned, or that you define the variable before using it to hold the variable pointer to an array variable. The following approach will always work correctly:

```
A = 0: A = VARPTR (A$ (0)).
```

VERIFY

Syntax

VERIFY *argument*

Purpose

This statement allows you to enable or disable Disk BASIC's sector write-verification feature. When VERIFY is on, every sector that is written to the disk as a result of statements such as SAVE, WRITE, PRINT, PUT, OPEN and DSKO$ is read back again to confirm that the data is accessible. The verification process does not actually compare the data written to the data read back. All it does is read the sector to ensure that the data can be read and that no CRC error occurs. Obviously, disk writes will require about twice as much time to complete when the VERIFY feature is active. When the computer is first turned on, VERIFY is OFF.

Arguments

argument must be either ON or OFF.

Potential Errors

None.

Examples

```
VERIFY ON

VERIFY OFF
```

WRITE

Syntax

WRITE *[#buffer,] data list*

Purpose

This statement is normally used to put data into a disk file buffer, but it can be used to send quoted strings to cassette, printer, or screen as well. WRITE differs from PRINT in that it automatically encloses *data list* in quotation marks. If a LINE INPUT statement is used to read back the data that was saved with a WRITE statement, the entire *data list* will be read as one item; therefore, INPUT should always be used when retrieving data saved with WRITE. If *buffer* is omitted, the data is written to the screen.

Arguments

buffer may be any numeric expression referring to an open file buffer between -2 and 15.

data list may be one or more string expressions or numeric expressions; multiple items must be separated by commas.

Potential Errors

DF - the last WRITE statement caused a buffer-full condition, which resulted in a new sector being created on the disk. When the disk write occurred, the disk was found to be full.

DN - *buffer* is out of range.

ER - you have attempted a WRITE to a direct (random) access disk file without having first performed a corresponding PUT.

IO - the disk is not properly inserted in the drive; the drive door is not closed; or a write error occurred.

NO - *buffer* does not refer to an open disk or cassette file.

VF - VERIFY is on; the data just written to disk cannot be read back.

WP - the disk is write-protected.

Examples

```
WRITE #2, A$, B, C$, D

WRITE # F, A
```

Section Two

PROGRAM OPTIMIZATION TECHNIQUES

PROGRAM OPTIMIZATION TECHNIQUES

The interpreter ROMs developed by Microsoft Corporation for the Color Computer provide a powerful and versatile implementation of the BASIC language. In order to make the language as versatile as possible, however, the developers were forced to make some sacrifices which resulted in certain limitations, particularly with respect to execution speed. The aim of this section is to assist you in making BASIC work faster and more efficiently. Unfortunately, for reasons which will become more obvious as we progress, some of the optimization techniques greatly reduce the readability of your program.

In order to demonstrate the various time-saving programming techniques, we have put together a series of sample programs that use the TIMER function to measure loop speeds. You will notice that the programs are quite trivial; the speed increases we achieved in these example programs will not necessarily be the same in longer programs. Consider, for example, the test programs that appear in the first sub-section, "Using REM Statements". You can easily see that Program B runs almost twice as fast as Program A. True, but remember that the subroutine consists only of the single statement RETURN. If, instead, the subroutine contained 30 or 40 statements of code, the speed increase would not be nearly as dramatic; nevertheless, Program B would still be faster that Program A.

USING REM STATEMENTS

REMs are used to add comments to a program; comments can be immensely helpful to you, especially when you return to a program at some (long) time after the program has been completed, either to make additions or changes, or to eliminate a recently discovered bug.

When developing a program, don't skimp on remarks. In fact, we recommend that you be as liberal as you need to be in order to make the program completely understandable to you. We also recommend that you place remarks in their own lines, preferably with unusual line numbers such as 17, 239, etc. This way, you can reserve line numbers that are multiples of 10 to be used by the legitimate code. Later on, when you decide to remove remarks to produce your working version of the program, you can be a little more certain that you are not deleting lines which are referenced elsewhere in your program.

REMs enhance program readability and help to produce "self-documenting code". But that is where their usefulness ends. REMs use up a great deal of memory and slow down the operation of a program, particularly when the interpreter has to scan the remark lines. The following two programs should illustrate this point:

```
--- Program A ---

00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  GOSUB 70
00040  NEXT T
00050  PRINT TIMER
00060  END
00070  REM SUBROUTINE
00080  RETURN

RUN

 412

OK
```

--- Program B ---

```
00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  GOSUB 80
00040  NEXT T
00050  PRINT TIMER
00060  END
00070  REM SUBROUTINE
00080  RETURN

RUN

 270

OK
```

The only difference between the two programs is that the first calls a REM, while the second calls the subroutine directly. The reason for the speed increase is that in Program A, line 70 contains a remark which must be interpreted. (In actual fact, the interpreter does not use the line link to skip over the remark; instead, it must scan the entire line to find the start of the next line.)

Another reason you should never GOTO or GOSUB to a REM line is that problems can occur when you use a utility program (such as BASMUNCH) at some later date to remove the REMs. Now you'll have to go through the program and change all the references to the deleted REMs.

Speaking of deleting lines, let's digress for a moment. Suppose you have deleted a number of lines from your program but you are unsure whether you have deleted a referenced line number. There's a way to get BASIC to let you know: simply RENUMber the program without changing any line numbers. What was that again? Well, here's the trick:

```
RENUM 63999, 63999
```

Although the program will not actually be renumbered, any referenced line numbers that have been deleted from the program will be listed as UL errors. The number 63999 is used because it is the highest line number allowed by BASIC. Since your program cannot have any legitimate line numbers above this number, no line numbers can change. In spite of this, the interpreter still goes through the motions and checks all line references.

In general, our approach is to be quite liberal with REMarks as we develop programs, making certain at all times that no GOTO, GOSUB or IF-THEN-ELSE statements make reference to a remark line. We usually accomplish this by reserving line numbers ending in 7, 8 or 9 for remarks and all other line numbers for executable code. When program development is complete, we retain a renumbered copy (or two or three) of the fully commented program, and use a utility like BASMUNCH (available separately) to compress the program. The compressed version then becomes our working copy of the program.

One last note about REMarks: we have found that typing comments in lower case (reverse video) makes them easier to find when scrolling through a listing on the screen.

SUBROUTINES

Subroutines can be a programmer's best friend. Not only do they eliminate the need to repeat blocks of code; they also permit the development of specialized routines that can be debugged, placed in a separate library, and quickly incorporated into new programs.

Where is the best location for a subroutine? Well, it depends on the situation. Subroutines that are used only occasionally, or those whose speed is not important, should be placed at the end of the program. Those used most frequently in a given program should be placed either right AFTER the code calling them, or at the beginning of the program.

When BASIC encounters a statement which makes reference to a new line number (i.e. GOTO, GOSUB, or IF-THEN-ELSE), the destination line number is compared with the line number containing the calling statement. If the new line is greater than the current one, the interpreter starts looking for the new line starting at the current position in the program; otherwise, it starts at the beginning of the program.

To illustate, we took a fairly long program (225 lines, 8K) and inserted the following code:

At the beginning of the program,

```
00001  RETURN
```

In the middle,

```
00099  RETURN
00100  TIMER = 0
      :FOR T = 1 TO 1000
      :GOSUB 104
      :NEXT T
      :PRINT "AFTER CALLING ROUTINE:  "; TIMER
00101  TIMER = 0
      :FOR T = 1 TO 1000
      :GOSUB 226
      :NEXT T
      :PRINT "AT END OF PROGRAM:      "; TIMER
00102  TIMER = 0
      :FOR T = 1 TO 1000
      :GOSUB 1
      :NEXT T
      :PRINT "AT BEGINNING OF PROGRAM:"; TIMER
```

```
00103  TIMER = 0
      :FOR T = 1 TO 1000
      :GOSUB 99
      :NEXT T
      :PRINT "BEFORE CALLING ROUTINE: "; TIMER
00104  RETURN
```

And at the end,

```
00226  RETURN

RUN 100

AFTER CALLING ROUTINE:   286
AT END OF PROGRAM:       506
AT BEGINNING OF PROGRAM: 262
BEFORE CALLING ROUTINE:  453

OK
```

The different speeds are a direct result of the time spent searching for the destination line numbers.

A little more food for thought: when a GOSUB statement is encountered, the return address is saved on the stack. Therefore, no line number search is performed when the RETURN statement is encountered. This makes a GOSUB/RETURN combination quite a bit faster than a similar construction involving two GOTO statements. Consider the two test programs below.

```
--- Program A ---

00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  GOSUB 70
00040  NEXT
00050  PRINT TIMER
00060  END
00070  RETURN

RUN

 256

OK
```

--- Program B ---

```
00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  GOTO 70
00040  NEXT
00050  PRINT TIMER
00060  END
00070  GOTO 40

RUN

 281

OK
```

Note that the placement of the calling statement will have a great effect on the timing in the above examples. For instance, if the calling statement were in line 200 instead of line 30, Program B would take even longer to execute.

Subroutines can save a great deal of memory. Use them, but give their placement some thought.

VARIABLES AND CONSTANTS

Color BASIC allows for two types of variables, namely simple and array variables. Each of these variable types may hold either string data or numeric data. Unlike other implementations of BASIC, Color BASIC does not permit options for specifying integer, or single- or double-precision variables. In the color computer, all numbers are stored as five-byte floating point values. On the other hand, Color BASIC allows you to define strings of from 0 to 255 characters in length, whereas many other versions of BASIC require you to dimension string sizes at the beginning of the program.

Variable names can be from 1 to 255 characters in length, but if the name is longer than one character, only the first two characters are actually used by BASIC. This means that the variables STREET, STATUS and STRAWBERRY would all be interpreted as ST. Long variable names can add a great deal to the readability of a program, but obviously each extra character that you put into a name uses an extra byte of program memory. A little less obvious is the fact that long names slow down program execution. Even though BASIC only recognizes the first two characters, the interpreter must still scan ALL the letters of the name--the longer the variable name, the longer this takes. For these reasons, we recommend that you use single-letter names for your most frequently used variables, and two-letter names for all others.

Although no optimization can be gained from this, we also suggest that you develop a convention for standardizing the variables that you use. For example, one of us always uses T and TT in FOR-NEXT loops and the other uses I and J. Which ones are used isn't really important, but the consistency is. When we look at our own programs, this approach helps us to know what certain variables mean right away. Another example of standardization is the use of the "O" variables in the subroutines in Sections 3 and 4 of this book. By reserving variables for specific uses, the possibility of duplication in a long program is reduced. When programming, you should also keep a written list of the variables you use. Try to use meaningful names: Both TE an PH seem reasonable for "telephone number", but AI doesn't.

Storage Of Simple Variables

All simple variables and function definitions are stored in memory in a "variable table" which immediately follows the end of the BASIC program. The table is expanded in size by the interpreter as each new variable is declared. The table is never

reduced in size except when a NEW or CLEAR statement is encountered, a program is RUN, or the computer is turned off (or a cold restart is performed), in which case all of BASIC's pointers are reset.

Within the table, all simple variables and functions have the same format. Each entry consists of seven bytes. The first two bytes are the "header block", which contains the variable name or the function name. If the name consists of only one character, then the second byte of the header block is cleared to zero. In a string variable entry, the most significant bit (bit 7) of the second byte of the header block is set to one. In a function definition entry, the most significant bit (bit 7) of the first byte of the header block is set to one. BASIC uses this "masking" technique to distinguish the different types of entries contained in the table.

The remaining five bytes of the variable entry are the "data block". In the case of a numeric variable, these five bytes are, in fact, the floating point value of the variable. In the case of a string variable, the data block contains a "string descriptor" that reveals the length of the string and where in memory it is actually located. In the case of a function definition entry, the data block contains a "function descriptor" that contains pointers to where the definition is located within your program and to where the parameter variable is located in the variable table. The charts below give a better picture of how the data is stored.

Numeric Variable Entry

Bytes	Purpose
0-1	variable name (header block)
2-6	floating point value (data block)

String Variable Entry

Bytes	Purpose
0-1	name--bit 7 of byte 1 set (header block)
2	length of string \
3	not used \
	) (data block)
4-5	address of string /
6	not used /

Function Definition Entry

Bytes	Purpose
0-1	name--bit 7 of byte 0 set (header block)
2-3	address of definition \
4-5	address of parameter) (data block)
6	not used /

Let's now consider how numbers are actually stored in the computer's memory. We mentioned earlier that all numeric data is stored in blocks of five bytes in what is known as "floating point format". This concept is best explained through the use of the more familiar scientific notation. Given the decimal number 123450.987, we can use scientific notation as an alternate representation, in any of the following forms:

123450.987E0
1234.50987E2
123450987E-3
1.23450987E5

In each of these examples, the string of digits to the left of the letter "E" is called the "mantissa", or the value of the number, while the digits to the right make up the "exponent", or the power of ten which must be applied to the mantissa in order to obtain the true magnitude of the number. If we accept the fourth example as being the standard format for all numbers represented in scientific notation, we can say that the number has been "normalized"; that is, the decimal point is shifted until there is only one digit to the left of it and for each shift the exponent is adjusted accordingly. For each shift to the left, the exponent is increased by one; for each shift to the right, the exponent is decreased by one. Since this is the case, it is not difficult to come to the conclusion that the exponent actually represents the number of times the decimal point was shifted. A positive exponent indicates that the decimal point was shifted to the left; a negative exponent indicates right shifts. From this, we can also conclude that to obtain the true magnitude of the number, all we have to do is shift the decimal point the indicated number of positions in the OPPOSITE direction, adding zeroes if necessary as we proceed.

Because the decimal point moves from one position to another, the term "floating point" is an apt description, and the approach just outlined is essentially how BASIC converts numeric data into the desired representation. In reality, the Color Computer stores all data as a series of binary digits, so we end up dealing with a "binary point" instead of a decimal

point, powers of two instead of powers of ten, and bits instead of decimal digits.

During the "normalizing" process, the interpreter actually stops when the most significant bit is immediately to the right of the binary point, instead of to the left. Then the most significant bit of the mantissa can be used as a sign bit, 0 being positive and 1 being negative. As far as the value of the mantissa is concerned, the most significant bit is always assumed to be 1, regardless of the physical status of the bit. In this way, BASIC is able to use a single bit position for two purposes. Just as our decimal exponents above represent the number of decimal point shifts, so does the binary exponent represent the number of binary point shifts. Once the binary exponent has been determined, however, the value &H80 (128 decimal) is added to it, thus giving us what is called a "biased exponent". As a result of the applied bias, any exponent in the range &H80 to &HFF (128 to 255) represents a positive exponent, while results in the range &H7F to &H01 (127 to 1) represent negative exponents. This actually translates to binary exponents in the range -127 to +127. Note that a stored exponent value of &H80 actually represents a value of zero. If the stored value is actually &H00, this indicates that the whole number is equal to zero.

Diagramatically, this is how the floating point data looks in memory:

Bytes	Purpose
0	biased exponent (number of shifts)
1	MS Byte of mantissa normalized until MS bit set, then MS bit becomes the sign bit
2-4	3 LS bytes of mantissa

What about the interpretation of floating point binary numbers? Although the task may at first appear to be monumental, it is really not that difficult, particularly if you approach the problem logically. Suppose we examine the data block of a floating point variable and find the hexadecimal numbers

83 C2 00 00 00

We know that the first byte in the series is the biased exponent. Since this byte is not zero, we know immediately that the whole number is a non-zero value, and that we must remove the bias by subtracting &H80. This subtraction leaves us with an exponent of +3, which tells us that the mantissa, when

deciphered, will have to be multiplied by 2^3, or 8.

Putting the calculated multiplier aside for the moment, let's turn our attention to the mantissa. The process of interpreting the mantissa is easier if we convert the hexadecimal value into its binary equivalent. We will ignore the least significant three bytes, since they are all equal to zero. The binary equivalent of the most significant byte is

11000010.

Remember that the most significant bit of this byte represents the sign of the mantissa. Since this bit is set to one, we know immediately that the mantissa is negative. We also know that the original floating point number was "normalized" to make the most significant bit equal to one, so having removed the sign bit and having noted its value, we immediately replace the sign bit with a one, which, in this case does not alter the appearance of the binary number:

1 (sign bit) 11000010 (normalized 1 bit replaced)

The next thing to remember is that the normalization process stopped when the bit immediately to the right of the binary point was set. This means that the string of bits above should actually be written as

.11000010

What exactly does this mean? If this were a decimal number, we would be able to arrive at the correct conclusion based on the knowledge that the first digit to the right of the decimal point represents 1/10 or $1 * 10^{-1}$, the second digit represents 1/100 or $1 * 10^{-2}$, the third digit represents 0/1000 or $0 * 10^{-3}$, and so on. Unfortunately, it is not a decimal fraction but rather a binary fraction. It is a simple case of replacing the powers of 10 just mentioned with corresponding powers of 2, with the following result:

$$1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 0 * 2^{-4} + 0 * 2^{-5} + 0 * 2^{-6} + 1 * 2^{-7} + 0 * 2^{-8}$$

This simplifies to:

$$1 * 1/2 + 1 * 1/4 + 1 * 1/128,$$

which further simplifies to

97/128

From our very first operation, we know that we have to multiply this number by 8, so we can quickly obtain the number

8 * 97/128 = 97/16 = 6.0625

The only thing left to do is to apply the value of the sign, which we know is negative. Thus, we know that

83 C2 00 00 00 = -6.0625

As you can see, the process of interpreting floating point binary data is not that difficult, although admittedly, it does become more complex when the second, third, and fourth bytes of the mantissa contain non-zero values. From left to right, the bits in each byte represent powers of -9 to -16, -17 to -24, and -25 to -32 respectively. Incidentally, the number +6.0625 would be converted to the floating point value

83 42 00 00 00

because the sign bit would be cleared to zero to indicate a positive number.

One of the limits of the four-byte mantissa is that the Color Computer cannot retain more than nine significant decimal digits. If you want to achieve a higher degree of accuracy, you must write your own double-precision (or higher) number-crunching routine. Below is a table that depicts the absolute limits on the range of displayable numbers.

POSITIVE

MINIMUM		MAXIMUM
01 00 00 00 00	<-- floating -->	FF 7F FF FF FF
+2.93873588E-39	<-- decimal -->	+1.70141183E+38

NEGATIVE

MINIMUM		MAXIMUM
01 80 00 00 00	<-- floating -->	FF FF FF FF FF
-2.93873588E-39	<-- decimal -->	-1.70141183E+38

Let's turn our attention now to the way in which string variables are structured. As we pointed out earlier, the data block of a string variable contains only three bytes which are of any use, namely bytes zero (the string length) and two and three (the string address). Since the string length is contained in only one byte, this means that a string can be from 0 to 255 characters in length. On the other hand, the address is stored

in two bytes, which means that the string can (theoretically) be located anywhere within the space of 64K of memory, from address &H0000 to &HFFFF. In fact, BASIC never uses a string address above &H7FFF because this area is reserved for your interpreter ROMs.

Suppose that your program contains the statements

```
00010  A$ = "HELLO THERE"
00020  B$ = A$
00030  C$ = A$ + ""
```

The string in line 10 is, in effect, a constant value which can be stored permanently in your program. When BASIC encounters this line, an entry will be created in the variable table such that the length byte is set to 11 and the address bytes point directly to the string in your program. In line 20, B$ is set equal to A$. This causes a new entry to be made in the variable table, but this time the contents of the data block for A$ are simply copied into the data block for B$; the end result is that both A$ and B$ point to exactly the same area of memory. Line 30 produces a slightly different result because C$ is a formulated string rather than a constant. Even though C$ contains exactly the same value as A$ because of the null string concatenation, the address bytes for C$ will point to a place near the top of your available RAM space, in a place called the "string pool". When the interpreter encounters line 30, it still creates an entry in the variable table, but this time it must first copy the contents of A$ into the string pool, perform the concatenation, and then assign the length and address attributes to the new variable.

If you want to examine the contents of a simple variable directly, you can do so by using the VARPTR function. Whether applied to a numeric variable or a string variable, VARPTR always returns a two-byte address (from 0 to 65535) which points to the start of the variable's data block. For example, if you obtained the VARPTR for variable A, the address would point at byte zero of the floating point number, i.e. its exponent. The remaining bytes of the number can be easily accessed by applying an offset of 1, 2, 3, or 4 to the obtained address. Similarly, if you want to examine the header block for a simple variable (the variable name) you can do so by applying an offset of -2 for the first character and -1 for the second character of the name.

Storage Of Array Variables

In principal, an array is nothing more than a collection (more specifically, a list) of related variables, each of which is accessed and/or modified by making reference to its "index". In order for Color BASIC to assign sufficient memory space to an array, the interpreter must know a few things about the structure: the name of the array, the number of dimensions, and the number of elements in each dimension. For fairly obvious reasons, the allocation of space for an array variable is a more complicated process than for a simple variable, and it is quite possible (and likely) that two different arrays will be allotted different amounts of memory.

Not too many people have difficulty visualizing an array consisting of from one to three dimensions. But the moment the fourth and subsequent dimensions are introduced, many people find themselves totally confused. In fact the visualization process is quite easy, if you work your way up from the simple to the complex. Suppose, for example, that you have a cash register in front of you. For the moment, let's assume that the register contains one drawer, and inside the drawer is a plastic tray containing five buckets oriented horizontally. This is a not-quite-typical cash register which can hold the quarters in one bucket, the dimes in the next, followed by the nickels and the pennies. The last bucket, of course, will hold our rolled coins. To identify any given bucket we need only give the bucket a name such as "A" or "Quarters" or "1". Having done that much, it is very easy to specify where to look for a specific kind of coin. In most computers, the name applied to such a "bucket" is called an index, and is usually an integer number ranging from zero to one less than the maximum number of "buckets". You should have no trouble realizing that the tray within the cash register drawer is, in reality, a one-dimensional array consisting of five "buckets" or "elements" numbered from zero to four.

Now suppose we add another tray to the drawer to hold the $20-dollar bills, the $10s, the $5s, the $2s, and the $1s respectively. This tray by itself represents another one-dimensional array containing five "buckets", again numbered from 0 to 4. With respect to the drawer, however, the second tray actually adds another dimension to our original array. Now, to specify where to look for a certain denomination of currency, we must specify not only the bucket number but also the tray number. Since we have only two trays, we will number them conventionally from 0 to 1. Thus if tray number 0 contains the

coin buckets and tray number 1 contains the bill buckets, in order to specify the location of the $20-dollar bills, we would have to say "put $20 in tray #1, bucket #0".

Up to this point, we have adhered pretty closely to the traditional concept of "two-dimensionality". We are about to depart from that narrow path, in order to simplify the concept of "multi-dimensionality". In the last example, we ended up with a two-dimensional array in which each of two "trays" contained 5 "buckets". If we give the array the name "A", we can use a BASIC statement to set aside enough memory for this array:

```
00010  DIM A (1,4)
```

The numbers "1" and "4" are used in the DIMension statement, instead of "2" and "5", because they represent the highest possible index values that can be assigned. BASIC knows that the lowest index value is zero in every dimension of the array, so sufficient memory will be allocated. (There is nothing wrong with using the values "2" and "5" if it makes you more comfortable counting from one upwards--just be aware of the fact that the zeroth element is also reserved by BASIC. As we will see later, this can be quite costly in terms of total memory allocation.)

Let's take our example a little further. In any supermarket, there may be from 10 to 20 separate checkout counters, each of which has its own cash register. (Already, a little light is beginning to shine!) That's right--now we must specify a "cash register" as well as a "tray" and a "bucket". As usual, we will number the registers from 0 to 9 (or 19 if you happen to be in a super-duper-market!) The BASIC statement that defines this "three-dimensional" array has to be modified:

```
00010  DIM A (9,1,4)
```

In most cities, supermarkets exist as "chains"; that is, generally, you will find more than one store location in your city. From store to store, the number of cash registers may change, but for this example, we will assume that each store has exactly 10 registers. (Is the light shining a little brighter?) You've got it--now, in addition to all the other indices, we must also be able to indicate which store we are referring to, and (assuming there are seven locations in your city) the BASIC DIM statement becomes:

```
00010  DIM A (6,9,1,4)
```

With this type of approach, it is quite easy to visualize a huge chain of stores around the world. To locate a certain denomination of currency, we would (theoretically) have to specify the continent, the country, the city, the store, the cash register, the tray, and the bucket--this is an array with seven dimensions! A DIM statement for such an array would look something like the following:

```
00010  DIM A (4,11,35,6,9,1,4)
```

It looks ominous, doesn't it? But it really isn't. It simply tells us that there are five continents (numbered from 0 to 4), each of which contains exactly 12 countries (numbered from 0 to 11). Within each country, there are 36 cities, each of which contains 7 supermarkets. Within each supermarket, you will find 10 cash registers, each of which consists of 2 trays containing 5 buckets each! Don't worry! You will not likely ever have to use an array of such a size. This particular array will require 7,560,019 bytes of memory! Although there are Color Computers in existence with 128K of RAM, we do not currently believe that 8 or 9 megabytes will ever become commonplace in the CoCo!

In each of the multi-dimensional examples above, we listed the indices in parentheses from left to right, where the leftmost index referred to the least specific array identifier (in this case, "continent") and the rightmost index referred to the most specific identifier (namely, "tray"). As far as the BASIC interpreter is concerned, our ordering convention is completely arbitrary, but from the point of view of being able to follow the logic of a given program, it makes sense to keep things orderly.

Now we address the question of how BASIC stores the data in memory. BASIC reserves an area immediately following the table of simple variables specifically for array storage. Just as the simple variable table can contain a mixture of numeric and string variables, so can the array table contain a mixture of numeric and string arrays, although each array occupies an area of memory which must be contiguous (that is, uninterrupted). As well, each array may hold only one data type, either numeric data or string data. Because of this requirement, an array consists of a header block, which contains information about the number of dimensions and the number of elements in each dimension, and a data block, which contains a series of 5-byte entities, each of which is either a floating point number or a string descriptor, depending on the array type.

In order to gain a better understanding of how an array is physically stored in memory, let us assume that we have a one-dimensional array named "A" which we have defined by the BASIC statement "DIM A (3)". If we examine the array table after declaring this array, we will find a series of hexadecimal values, as follows:

```
41 00                 * array name (A)
00 1B                 * offset to next array n
01                    * number of dimensions
00 04                 * number of elements in first dimension
00 00 00 00 00        * data element #0
00 00 00 00 00        * data element #1
00 00 00 00 00        * data element #2
00 00 00 00 00        * data element #3
```

The data block for this array is 20 bytes long, which makes sense since we specified 4 elements, each of which occupies 5 bytes of memory. Notice that the value of each element was automatically set to zero during the execution of the DIM statement. The header block is a little more interesting. As with simple variables, there is space reserved for the name of the array (bytes #0 and #1). But in addition to these two bytes, five additional bytes are reserved. Bytes #2 and #3 represent a count of the total number of memory locations occupied by the current array; these bytes are used only when BASIC is searching through the table for a specific array and are necessary because of the fact that no two arrays need necessarily occupy the same amount of space. Byte #4 tells BASIC how many dimensions are contained in this array. Since it is only a one-byte value, this means that it is theoretically possible to have an array of from 1 to 255 dimensions! (However, because of the limitations imposed on the size of a BASIC program line, it is virtually impossible to specify such a large array.) Following the dimension specifier, there will be a pair of bytes for *each dimension* which tell BASIC how many elements are in each dimension. For our single-dimension array example, there are only two additional bytes (#5 and #6) and these tell us that the array consists of only four elements.

Now take a look at a similar array, A$(3):

```
41 80                  * array name (A$)
00 1B                  * offset to next array
01                     * number of dimensions
00 04                  * number of elements in first dimension
00 00 00 00 00         * descriptor element #0
00 00 00 00 00         * descriptor element #1
00 00 00 00 00         * descriptor element #2
00 00 00 00 00         * descriptor element #3
```

Notice that the only difference between this and the previous array is that the name portion of the header contains the value "80" which indicates that the array is a list of string descriptors. This, you will recall, is exactly the same convention that is used to differentiate between simple numeric and string variables. Notice, too, that each of the descriptors has been set to all zeros. This makes sense if you remember that when a string variable is DIMensioned, it is set equal to the null string, which has a length of zero. Since its length is zero, it doesn't really matter where the address pointer actually points. The writers of BASIC chose to simplify their task by making all of the descriptor bytes equal to zero.

Now, let's skip a few steps and examine the storage of a small, three-dimensional array which we have initialized as follows:

```
00010  DIM I, J, K, X
00020  DIM A (0, 1, 2)
00030  FOR I = 0 TO 0
      :FOR J = 0 TO 1
      :FOR K = 0 TO 2
00040  A (I, J, K) = X
      :X = X + 1
00050  NEXT K, J, I
```

If you examine the logic of this short program, you will see that we have performed the initialization using "normal odometer" format. What this means is that the elements of the array contain the following values:

```
A (0, 0, 0) = 0
A (0, 0, 1) = 1
A (0, 0, 2) = 2
A (0, 1, 0) = 3
A (0, 1, 1) = 4
A (0, 1, 2) = 5
```

Notice that the rightmost index increases in value first followed by the next index to the left, and so on, which is exactly how the odometer in your car works.

When we examine memory following this setup, however, we are in for a mild surprise:

```
41 00                  * array name (A)
00 29                  * offset to next array
03                     * number of dimensions
00 03                  * number of elements in last dimension
00 02                  * number of elements in middle dimension
00 01                  * number of elements in first dimension
00 00 00 00 00         * data element 0,0,0 (value 0)
82 40 00 00 00         * data element 0,1,0 (value 3)
81 00 00 00 00         * data element 0,0,1 (value 1)
83 00 00 00 00         * data element 0,1,1 (value 4)
82 00 00 00 00         * data element 0,0,2 (value 2)
83 20 00 00 00         * data element 0,1,2 (value 5)
```

There are a couple of interesting points to note in the memory layout for this array. The first point is the header block has been increased in size to allow for the specification of the number of elements in each dimension. This in itself is not particularly startling. What is noteworthy is the fact that the dimension size bytes are stored in the opposite order to which they were listed in the DIM statement. The second point is that the floating-point values are stored in what is called "reverse odometer" format (the left-most index increases first, followed by the next-to-left index, and so on), which is quite different from the way in which the array was initialized. This reversal of data storage does not make a whole lot of sense until you consider the steps required for BASIC to locate a specific array element.

Let's consider how BASIC would locate an element in a single-dimension array. This is a fairly trivial case: all the interpreter has to do is multiply the index value by five (since each element of the array actually occupies five bytes of memory) and add in the known address of the first element (element #0) and the location of the desired value is immediately known.

In a two-dimensional array, the problem is still quite simple. Suppose we have declared array A as follows:

```
00010  DIM A (1, 3)
```

From this statement we know that the first dimension contains two elements (#0 and #1), and the second dimension contains four elements. Suppose further that at some point in the program we make reference to element A (1, 0). How does BASIC know where to look for that element? The interpreter knows where the array table starts; with this start point, BASIC searches forward until it finds the header block for array A. (If the array is never found, BASIC quickly performs a default DIM statement, setting aside 11 elements in each dimension.) Now a check is made to ensure that the element specified can in fact exist--this check verifies that there are exactly two dimensions in the array and that the subscripts are within limits. If the check succeeds, then BASIC knows the address of element A (0, 0). In order to determine the offset of the desired element from the base address, a calculation must be performed, as follows:

*offset = 5 * (1 + 0 * 4).*

The "5" represents the constant size of each data element; the "1" represents the value of the left subscript; the "0" represents the value of the right subscript; and the "4" represents the number of elements in the second dimension. In general, a two-dimensional array contains J elements in one dimension and K elements in the other. If we attempt to access element (a, b) in the array, the above equation can be re-written as

*offset = 5 * (a + b * K).*

If we have a three-dimensional array containing J, K, and L elements in each dimension respectively, and we wish to access element (a, b, c), the equation becomes a little more formidable:

*offset = 5 * (a + b * K + c * L * K).*

This may be a little difficult for you to accept, but if you try a few examples, you will see that it is correct. It is beyond the scope of this text to include a proof of this equation; suffice to say that if you consider the example of a simple cube, you will begin to understand the need for all the multiplications.

In actual fact the multiplications are performed in the reverse order to that specified above, so that, in reality, the equation looks as follows:

$$offset = 5 * (c * L * K + b * K + a)$$

and this, of course, explains why the data is physically stored in memory in reverse order to the way in which it is specified.

In general, if we have an N-dimensional array with A_1 elements in dimension #1, A_2 elements in dimension #2, and so on, and we wish to access element $(J_1, J_2, \ldots, J_N)$, the formula required to locate the data is:

$$\begin{aligned} offset = 5 * ((&J_N * A_N * A_{N-1} * \ldots * A_2) + \\ &(J_{N-1} * A_{N-1} * A_{N-2} * \ldots * A_2) + \\ &(J_{N-2} * A_{N-2} * A_{N-3} * \ldots * A_2) + \\ &\vdots \\ &(J_3 * A_3 * A_2) + \\ &(J_2 * A_2) + \\ &(A_1)) \end{aligned}$$

Our only purpose in writing out this rather ominous looking formula is to emphasize the fact that each additional dimension in an array causes the interpreter to go through an additional level of arithmetic before it can locate a specific element in the array. This extra level of arithmetic can be quite time consuming, and therefore it makes sense to use multiple single-dimensional arrays instead of one multi-dimensional array. For example, suppose you want to implement a two-dimensional array, in which one dimension represents a person's age and the other dimension represents his/her salary. Obviously, this could be easily implemented as an array such as A(1,10). But is could also be implemented as two one-dimensional arrays such as A(10) and B(10). Now suppose you wanted to obtain both the age and the salary of the fourth person (element #3) in the array. In the case of the two-dimensional array, you would have to access both dimensions of the array (i.e., age comes from A(0,3) and salary comes from A(1,3)), while in the case of the one-dimensional arrays, you would have to access each array separately (i.e., age comes from A(3) and salary comes from B(3)).

Using the system of multiple single-dimensional arrays causes more memory to be used for storage of the program and for storage of the array information in the array variable table, but the access time is much faster. Consider the following programs:

```
--- Program A ---

00010  DIM A(10), B(10)
00020  FOR I = 0 TO 10
      :A(I) = I
      :B(I) = I
      :NEXT
00030  TIMER = 0
00040  FOR I = 1 TO 1000
00050  A = A(3)
      :B = B(3)
00060  NEXT
00070  PRINT TIMER

RUN

 607

OK

--- Program B ---


00010  DIM A(1,10)
00020  FOR I = 0 TO 1
      :FOR J = 0 TO 10
      :A(I, J) = I
      :NEXT J, I
00030  TIMER = 0
00040  FOR I = 1 TO 1000
00050  A = A(0, 3)
      :B = A(1, 3)
00060  NEXT
00070  PRINT TIMER

RUN

 913

OK
```

The first program, involving single-dimension arrays, runs almost 30 percent faster than the second program, and each of these programs does nothing more than make access to two separate array elements! The lesson is clear: avoid the use of arrays if possible, but if you must use them in your program, make every effort to stick with single-dimension arrays. At the very least, do not use an array variable when a simple variable will do--

```
--- Program A ---

00010  CLEAR 1000
00020  TIMER = 0
00030  FOR T = 1 TO 255
00040  A$ = A$ + CHR$ (T)
00050  NEXT T
00060  PRINT TIMER

RUN

 153

OK


--- Program B ---


00010  CLEAR 1000
00015  DIM A$(0)
00020  TIMER = 0
00030  FOR T = 1 TO 255
00040  A$(0) = A$(0) + CHR$ (T)
00050  NEXT T
00060  PRINT TIMER

RUN

 207

OK
```

As with simple variables, you can use the VARPTR function to obtain the address of any array element. The value obtained always points to byte zero of the five-byte data block for the specified element. In the case of a numeric array, this will of course be the exponent of a floating-point number; in the case of a string array, byte zero represents the length byte of the string. Each of the other bytes in the data block can be accessed by applying an offset of 1, 2, 3, or 4 to the obtained result. If you want to look at the header block of an array by using the VARPTR function, you must first obtain the address of the first element in the array (element zero). Then you can apply a negative offset whose size will be determined by the number of dimensions in the array. In general, the header will be five bytes plus two bytes for each dimension in the array. Thus, for a single-dimension array, the header will start seven bytes before the data block for element zero; for a two-dimension array, nine bytes before the data block; and so on.

You can easily determine the total number of elements in an array by adding one to each of the maximum index values and then multiplying each of the indices together. Thus, if you declare the array A(3, 9), the array will contain 4 x 10, or 40 elements.

Since each element occupies five bytes of memory, the following formula can be used to determine the total number of bytes that a DIM statement will reserve for any size array:

SIZE = 5 + (2 x number of dimensions) + (5 x total number of elements)

Table Lookup And Variable Positioning

Now that we have some understanding of the variable tables, let's look at what happens when a variable is encountered in a BASIC program.

First, it is determined if the variable is a simple variable or an array variable and the appropriate table is scanned. If the variable is found, its value (or address if it is a string) is accessed or updated, and the program continues. But what happens if the variable is not found in the table? If it is a simple variable, the name is added to the end of the simple variable table. But because the array table immediately follows the simple variable table in memory, the array table first has to be moved up 7 bytes to make room for the new simple variable. On the other hand, if the encountered variable happens to be an array variable that has not been previously dimensioned in a DIM statement, it is given a default DIM value of 10 for each dimension (i.e., 11 elements) and an appropriate entry is added to the end of the array variable table.

Two facts are important in the above. First, the tables are scanned each time a variable name is encountered. Second, variables are added to the tables as they are encountered in the program. The process of adding a new simple variable to the table can be quite time consuming, especially if large arrays have been dimensioned beforehand.

```
--- Program A ---

00010  TIMER = 0
00020  A = 1
00030  PRINT TIMER

RUN

 0

OK
```

--- Program B ---

```
00005  DIM A(2000)
00010  TIMER = 0
00020  A = 1
00030  PRINT TIMER

RUN

 14

OK
```

The difference in the execution speed of the two programs (about 1/4 second) is the time required to move the 10,007 byte table to make room for the variable "A".

The location of a variable relative to other variables in a program will also affect the execution speed of the program, particularly during the first run.

--- Program A ---

```
00010  A = 0
      :B = 0
      :C = 0
      :D = 0
      :E = 0
      :F = 0
      :G = 0
      :H = 0
      :I = 0
00015  XX = 0
00020  TIMER = 0
00030  FOR T = 1 TO 1000
00040  XX = XX + 1
00050  NEXT T
00060  PRINT TIMER

RUN

 380

OK
```

--- Program B ---

```
00005  XX = 0
00010  A = 0
      :B = 0
      :C = 0
      :D = 0
      :E = 0
      :F = 0
      :G = 0
      :H = 0
      :I = 0
00020  TIMER = 0
00030  FOR T = 1 TO 1000
00040  XX = XX + 1
00050  NEXT T
00060  PRINT TIMER

RUN

 333

OK
```

Even in the simple programs above, with their very short variable tables, the difference in the amount of time spent searching for the variable "XX" is evident. In Program A, "XX" will be the tenth entry in the variable table; in Program B, it is the first entry.

From these examples, we can conclude that often-used variables should be placed as close as possible to the beginning of the variable table; furthermore, because the array table has to be moved every time a new simple variable is encountered, it makes sense to DIMension all simple variables before arrays are DIMensioned. Note that:

```
00010  DIM A, B, C, XX, Z1$
```

is a perfectly legal statement; DIM needn't only be used to dimension arrays.

Declared Constants

Let's leave variables for a moment and discuss constants. BASIC does not understand a number like 456 or -12345.67. It must first be converted into a five-byte floating point representation before the program can continue. On the other hand, once a variable name is found in the array table, no conversion is necessary since the number is already stored in floating point format. Because no conversion is necessary, it follows that BASIC can process a declared constant much faster than it can process a number such as 456 or -12345.67.

```
--- Program A ---

00010  TIMER = 0
00020  FOR I = 1 TO 1000
00030  A = SIN(3.14159)
00040  NEXT
00050  PRINT TIMER

RUN

 3560

OK


--- Program B ---


00005  PI = 3.14159
00010  TIMER = 0
00020  FOR I = 1 TO 1000
00030  A = SIN(PI)
00040  NEXT
00050  PRINT TIMER

RUN

 2073

OK
```

Just by declaring the value 3.14159 as a constant, PI, we obtained more than a 40% increase in speed!

If the constant you are using happens to be an integer value in the range 0 to 65535, another option is to use "&H" notation. BASIC is able to convert the number &HFF to floating point faster than it can convert the number 255, but this method is still not as fast as using a declared constant.

```
--- Program A ---

00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  A = A + 255
00040  NEXT
00050  PRINT TIMER

RUN

 490

OK

--- Program B ---


00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  A = A + &HFF
00040  NEXT
00050  PRINT TIMER

RUN

 334

OK
```

--- Program C ---

```
00005  B = 255
00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  A = A + B
00040  NEXT
00050  PRINT TIMER

RUN

 312

OK
```

The trick of using hexadecimal notation can also be used when loading machine language routines from DATA statements. Not only does the data load more quickly (as expected, according to the results of the last example), but it is much easier to relate hexadecimal numbers back to the original assembly language source code. Unfortunately, the increase in speed and readability is partially nullified by the fact that hexadecimal notation uses quite a bit more memory.

--- Program A ---

```
00010  DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
       15, 16, 17, 18, 19, 20
00020  TIMER = 0
00030  FOR T = 1 TO 20
00040  RESTORE
00050  FOR TT = 1 TO 20
00060  READ A
00070  NEXT T, TT
00080  PRINT TIMER

RUN

 130

OK
```

```
--- Program B ---

00010  DATA &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9,
       &HA, &HB, &HC, &HD, &HE, &HF, &H10, &H11, &H12,
       &H13, &H14
00020  TIMER = 0
00030  FOR T = 1 TO 20
00040  RESTORE
00050  FOR TT = 1 TO 20
00060  READ A
00070  NEXT T, TT
00080  PRINT TIMER

RUN

 119

OK
```

Variables In FOR/NEXT Loops

Another use of variables is as counters in FOR/NEXT loops. The variable name is always required immediately after the FOR statement, but can be omitted after the NEXT statement--and this translates to savings in both memory requirements and execution speed. In a nested loop, "NEXT:NEXT" uses three bytes of memory; "NEXTA,B" uses four; and "NEXT AA, BB" uses eight (including the two spaces). Yet all three notations serve the the same purpose. The last format is easier to read while the first format is both quicker and more efficient. Again a short example will illustrate the point:

```
--- Program A ---

00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  NEXT T
00040  PRINT TIMER

RUN

 120
```

--- Program B ---

```
00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  NEXT
00040  PRINT TIMER

RUN

 105

OK
```

A 13% speed increase simply by leaving out the variable name in line 30!

Variable Summary

As you can see, there is quite a bit to be learned from studying the various idiosyncracies of variable storage. This is not to suggest that there is anything really complex about variables, just that there are many things to remember. It all boils down to a fairly simple, logical approach as you develop your programs: think about how much space each variable name will occupy in a program line, the order in which variables should be dimensioned, where and when variables should be defined as constant values, that sort of thing.

If you do consider all of the pertinent factors, you will find your BASIC programs running much more efficiently.

MACHINE LANGUAGE

Sometimes even the most efficient BASIC program will not run fast enough. This is particularly true in applications involving data sorts and/or graphic displays. Whereas the difference between an efficient and inefficient BASIC program may be in the order of 30 to 40 percent, machine language programs will often execute 30 to 100 TIMES faster than BASIC. We will not attempt to teach you assembly language in this book, but some comments on using machine language within a BASIC progrm are in order.

Marrying BASIC and machine language together can often be a frustrating experience. Two questions must be answered by the programmer: First, where to place the routine? Second, how to pass arguments between BASIC and machine language?

Machine Language Placement

Below are several areas that you may wish to consider for storing one or more machine language subroutines.

1. The Cassette Buffer

On power up, BASIC reserves a 256-byte buffer at &H01DA for cassette operations and a 9-byte buffer at &H01D1 for cassette file names. Since these 265 bytes are used only for cassette operations, your program can use them for a machine language subroutine. But be careful: any cassette operation will destroy the subroutine.

To place the routine in the cassette buffer you can either POKE it in from your BASIC program, or if you have disk drives, assemble the program at &H01D1 and perform a LOADM from within your BASIC program.

2. System RAM

BASIC uses the area from &H0000 to &H0400 for most of its house-keeping functions. Some locations are not used. There are 13 unused variable bytes at &H00F3. These could be used for a very short routine or for data storage. And if memory is at a premium and none of Extended BASIC's graphic functions are being used, you could consider using the graphic variable area from &H00B2 to &H00CA. Be very careful if you elect to use these areas--you may create a problem that is nearly impossible to find using normal debugging techniques.

3. Within A BASIC Program

One area for short machine language programs is right within your BASIC program, embedded in a series of strings. This technique, referred to as "string packing," can also be used to store graphics data.

In the following example, we will pack a 44 byte machine language program into a string called M$. To pack data into a string, first create a dummy string in the program with the same length as the number of data bytes to be stored.

```
00010  M$ = "12345678901234567890123456789012345678901234"
```

Next, find the start address of the dummy data by means of the VARPTR command.

```
00020  A = VARPTR (M$)
      :A = PEEK (A + 2) * 256 + PEEK (A + 3)
```

Last, POKE the required data values into successive locations within the string.

```
00030  FOR T = 0 TO 43
      :READ D
      :POKE T + A, D
      :NEXT
00040  DATA 1, 2, 3, 4 . . .
```

Once the string has been re-defined, the line that performs the changes and the line(s) containing the data (lines 30 and 40 above) can be deleted from the program. By placing an "EXEC A" instruction somewhere in your BASIC program following line 20, you will cause the machine language routine to be executed.

There are a number of drawbacks to this technique: First, &H00's and &H22's can not be contained in the packed string. An &H00 would be interpreted as an end-of-line marker, while an &H22 would be interpreted as a quotation mark. Second, if you renumber the program after the string has been packed, strange things may happen if it contains any of the following values: &H01, &H02, &H03, &HA5 (GOTO token), or &HA6 (GOSUB token). Third, if you save the program in ASCII format, any values greater than $7F (which are treated as BASIC tokens by the ASCII SAVE command) will be untokenized (i.e., converted to their fully written format). This, quite obviously will render your routine totally useless.

Instead of using "string packing", you may want to use a technique known as "REM packing". The technique for this is essentially the same as that for string packing: Create a dummy REM statement of the proper length, find its start address and POKE the actual data into successive memory locations.

To find the address of the REM statement, you can either use a monitor or use a line of code similar to the following:

```
00010  A = PEEK (&H2F) * 256 + PEEK (&H30)
       :REM**********
00020  IF PEEK (A) <> ASC("*") THEN A = A + 1
       :GOTO 20
```

Line 10, which contains the REM statement, passes the start address of the current program line to the variable A. Line 20 scans the line for the first "*". Once the first asterisk is found, subsequent lines can POKE the desired values into the REM statement, in the same way that string packing is performed. Once the packing is accomplished, the lines which perform the packing can be safely deleted.

All of the cautions mentioned in the discussion on string packing apply to REM packing. The only real advantage that REM packing has over string packing is that you can include &H22s in the REM line.

4. At the end of a BASIC program.

One of our favorite areas for machine languge routines is at the end of a BASIC program. Routines which are appended in this manner do not need to be POKEd in every time the program is run; furthermore, they are invisible to the user and they are safe from BASIC routines like RENUM.

This technique is possible because of the manner in which BASIC keeps track of the location of a program. The start address of your program is stored at locations &H0019 and &H001A, while the end address is stored at locations &H001B and &H001C. These addresses are used whenever a program is SAVEd, LISTed or RUN. They also allow the interpreter to determine which areas are available for variable tables, etc. But note that when a program is RUN or LISTed, the end address is not really used at all. Instead, execution of the selected command continues until three consecutive zero bytes are encountered.

To add a machine language routine to the end of a BASIC program, you must first determine the end address of the program. This can be accomplished by the following PEEK statement:

PEEK (&H1B) * 256 + PEEK (&H1C)

Once you have determined the end address, you can POKE your machine-language routine into successive memory locations starting at this address. When that task is complete, then you must modify the contents of the above addresses (&H001B and &H001C) to point one byte past the end of your routine. This will effectively make BASIC believe that your routine is part of the BASIC program. Now all you have to do is SAVE or CSAVE the modified program. When you subsequently reload the program the routine will still be intact.

In order to be able to execute the subroutine, your BASIC program will have to include a line which calculates the start address of the routine. The calculation is very simple--the start address of the routine is equal to the new end address of the BASIC program minus the length (in bytes) of the routine. Once the start address is known, you can set up a DEF USR statement or you can use the EXEC command to execute the routine.

The routine will move up and down in memory as lines are added or deleted from the BASIC program. For this reason, it is quite important that the routine be written in position-independent code. Remember, your routine will be (C)SAVEd and (C)LOADed right along with the BASIC program. But beware: if you attempt to save the program using the ASCII format, the appended routine will be lost.

To assist you in appending machine language routines to BASIC programs, we have included a special utility, MLAPPEND, on the program tape you can order by using the form at the back of this text.

5. Graphics Display Memory.

If your application does not require any of BASIC's extended graphic functions, then the memory that is reserved for graphics display can safely be used as a storage area for machine-language routines. As before, routines can be POKEd or (C)LOADMed into this memory area.

Just remember that if you decide to use graphic memory for your routines, you must avoid the use of all graphic commands, or else you run the risk of completely destroying the code.

6. At the end of RAM.

This is the area recommended in the BASIC reference manuals for machine language routines. Two steps must be followed: first a CLEAR statement must reserve sufficient memory, then the routine must be POKEd into memory.

Calling Routines and Passing Parameters

Machine language routines can be called from BASIC by either an EXEC statement or a USR function. Unless you are passing an argument with the USR function, use the EXEC statement--it is much faster. The following programs, both of which call an RTS instruction, show the difference in speed between the two BASIC commands:

```
--- Program A ---

00010  POKE &H01DA, &H39
00020  DEFUSR 0 = &H01DA
00030  TIMER = 0
00040  FOR T = 1 TO 1000
00050  A = USR 0 (0)
00060  NEXT T
00070  PRINT TIMER

RUN

 304

OK
```

```
--- Program B ---

00010  POKE &H01DA, &H39
00020  A = &H01DA
00030  TIMER = 0
00040  FOR T = 1 TO 1000
00050  EXEC A
00060  NEXT T
00070  PRINT TIMER

RUN

 126

OK
```

The first EXEC statement in a program must always be followed by an address argument. But subsequent EXECs do not need the address specified. The last address used becomes the default.

There are a number of ways that arguments can be passed back and forth between BASIC and machine language routines. The easiest (and quickest) way is to use the USR function (for details, see USR and VARPTR in section 1). By using a combination of USR and VARPTR, a string containing a list of arguments could be passed from BASIC. The machine language routine could them modify the string as necessary before returning to BASIC.

Another method, which happens to be quite slow, is to POKE values into a reserved area of memory. These values could then be used and/or modified by the machine lanuage routine and PEEKed by the BASIC program following the return.

DEFINED FUNCTIONS

Extended Color BASIC allows you to use the DEF FN command to define a function of a single variable. This feature can be quite useful in any program that requires similar complex calculations to be carried out in several different areas of the program. The DEF FN command will generally save you a fair bit of memory if it is used frequently in the program. But that is where its advantage ends. If execution speed is of importance, then don't use it--instead, replace all function calls with in-line code, which is about 20% faster.

As another alternative, consider the use of a subroutine. You may not get a dramatic increase in speed (in most cases, a subroutine is actually slightly slower than the FN statement) but you will get almost the same savings in memory space.

```
--- Program A ---

00010  DEF FN A(X) = SIN(X)
00020  TIMER = 0
00030  FOR I = 1 TO 1000
00040  B = FN A(3.14159)
00050  NEXT
00060  PRINT TIMER
      :END

RUN

 3673

OK

--- Program B ---


00020  TIMER = 0
00030  FOR I = 1 TO 1000
00040  GOSUB 70
00050  NEXT
```

```
00060  PRINT TIMER
      :END
00070  B = SIN(3.14159)
      :RETURN

RUN

 3728

OK
```

--- Program C ---

```
00020  TIMER = 0
00030  FOR I = 1 TO 1000
00040  B = SIN(3.14159)
00050  NEXT
00060  PRINT TIMER
      :END

RUN

 3558

OK
```

The actual speed differences will depend on the placement of the subroutine, and on the complexity of the calculations being performed. Remember, too, that you can do a lot more with subroutines than you can with function calls; you are not limited to performing calculations.

MULTIPLE STATEMENT LINES

You can save quite a bit of memory and improve the execution speed of your program, just by placing consecutive statements on the same program line instead of putting a single statement on each line. Obviously, this approach results in virtually unreadable code, but remember, this section is devoted to a discussion of program optimization, not beautification!

Each program line you create carries with it five bytes of overhead: two for the line number, two for the line link, and an end of line marker. On the other hand, a colon separating two statements is only one byte long. This means that each line you eliminate by cramming statements on the previous line saves you four bytes of memory space. In a long program, that savings can convert to several hundred bytes.

```
--- Program A ---

00010  TIMER = 0
00020  FOR T = 1 TO 1000
00030  A = A + 44
00040  NEXT T
00050  PRINT TIMER

RUN

 723

OK


--- Program B ---


00010  TIMER = 0
      :FOR T = 1 TO 1000
      :A = A + 44
      :NEXT T
      :PRINT TIMER

RUN

 702

OK
```

Not a substantial speed increase, we admit. Remember that the interpreter has to scan everything within your program, and this includes things which are not visible such as the line link and the end-of-line marker. If you remove unnecessary bytes from your program by putting as much as you can on a single line, then the scanning process can be executed much more quickly, and this is where the speed increase comes from.

The BASMUNCH utility (available separately) is able to compress BASIC programs and thereby save a great deal of space by creating multiple-statement program lines (some of which are longer than the maximum 254 bytes you can type in from the keyboard.)

COMMAND SELECTION

There are as many ways to write any single program as there are different programs to write! But only one or two selected approaches will produce the "best" program; i.e. the program with the best execution speed and minimum memory requirements.

Often, by using the fastest option or command available, great overall speed increases can be attained. For one example, consult our discussion of the "G" option for GET/PUT in section one.

In the following examples, Program A demonstrates one method of selecting a subroutine from a menu. Program B, which uses the ELSE statement instead of multiple lines, is a slight improvement, but Program C, with its use of INSTR and ON GOSUB, is nearly twice as fast and much more concise.

--- Program A ---

```
00010  TIMER = 0
00020  A$ = "D"
00030  FOR T = 1 TO 1000
00040  IF A$ = "A" THEN GOSUB 110
00050  IF A$ = "B" THEN GOSUB 120
00060  IF A$ = "C" THEN GOSUB 130
00070  IF A$ = "D" THEN GOSUB 140
00080  NEXT T
00090  PRINT TIMER
00100  END
00110  RETURN
00120  RETURN
00130  RETURN
00140  RETURN

RUN

 1132

OK
```

--- Program B ---

```
00010  TIMER = 0
00020  A$ = "D"
00030  FOR T = 1 TO 1000
00040  IF A$ = "A" THEN GOSUB 110
      :ELSE IF A$ = "B" THEN GOSUB 120
      :ELSE IF A$ = "C" THEN GOSUB 130
      :ELSE IF A$ = "D" THEN GOSUB 140
00080  NEXT T
00090  PRINT TIMER
00100  END
00110  RETURN
00120  RETURN
00130  RETURN
00140  RETURN

RUN

 1002

OK
```

--- Program C ---

```
00010  TIMER = 0
00020  A$ = "D"
00030  FOR T = 1 TO 1000
00040  ON INSTR ("ABCD", A$) GOSUB 110, 120, 130, 140
00080  NEXT T
00090  PRINT TIMER
00100  END
00110  RETURN
00120  RETURN
00130  RETURN
00140  RETURN

RUN

 595

OK
```

SPACES

Spaces between functions and variables undoubtedly make a program easier to read, but they consume valuable memory. Not only that, each space must be interpreted by BASIC, and this wastes valuable time.

Spacing is required only when a BASIC keyword directly follows a variable name or when two simple numeric variables occur one after the other.

The following examples show cases when no spacing is necessary:

1. PRINTAB
   "$" functions as the delimeter.

2. IFA(3)THENGOTO40
   ")" functions as the delimeter.

3. IFT$=Z$THEN40ELSEIFY=H(8)GOSUB100
   ")" and "$" function as delimiters.

But the spaces ARE required in the following examples:

4. PRINTA B
   If no space was included, BASIC would assume that you were referring to the variable "AB".

5. IFA GOTO40
   Without the space, BASIC would assume the variable "AGOTO40"

6. IFT=Z THEN40ELSEIFY=H GOSUB100
   Without any spaces the variable after the first "=" would be "ZTHEN40ELSEIFY", and after the second, "HGOSUB100."

The problem is not really with BASIC's interpretation routines, but rather with the routine that tokenizes the line when it is input from the keyboard. Once the keywords have been tokenized, the spaces can be safely deleted by a machine language utility like BASMUNCH.

COLONS

Colons are used to separate multiple statements on one line. They are not required in front of "ELSE" or "'" (the short form for REM). When BASIC tokenizes either of these statements, it automatically inserts an invisible colon in front of the keyword. Were you to put a colon in when typing the line, you would end up with two of them. The colons inserted by the tokenize routine do not show up during a LIST, but you can see them if you use a monitor program.

SUMMARY

In this section we have attempted to show you some tricks that will save memory and speed up program execution. By using all the techniques presented here, you should easily be able to achieve speed increases in the order of 10 to 30% over a "sloppy" programming style. Similar gains in memory usage should also be achieved. Here is a summary of our suggestions:

1. Avoid the use of REMs within your program; certainly, if you must include them, make sure that no part of your program makes a direct call to a line containing only a REMark.

2. Use subroutines for sections of code that must be used frequently. This will save quite a bit of memory. In order to make the subroutine run as fast as possible, try to place it as close as possible to the beginning of the program.

3. DIMension ALL variables. In particular, declare the simple variables first, followed by array variables, ordered according to frequency of use within the program. You can obtain a very good frequency count by using the utilities VARCOUNT, VARXREF and LINEXREF (available on the program tape you can order with the form supplied at the back of this book).

4. Avoid the use of array variables whenever simple variables will do. If you must use arrays, try to stick with single dimension arrays. Above all, always DIMension the array to the exact size needed; that way, you will be reserving only the amount of memory that you actually need.

5. Declare ALL constants at the beginning of your program; it will run much more quickly if you refer to a constant's name (e.g. PI) rather than to its value (e.g. 3.14159).

6. Do not be afraid to marry machine-language routines into your BASIC program. Machine language executes from 50 to 100 times faster than the equivalent BASIC instructions; furthermore, machine language enables you to do things which simply cannot be done in BASIC. Use the MLAPPEND utility to append the routine(s) to the end of your BASIC program.

7. Remember that EXEC is a faster instruction than its USR equivalent. Use EXEC whenever you wish to call a machine language routine without passing any parameters.

8. Avoid using DEF FN in your program; direct in-line code is much faster.

9. Place as many statements as possible on the same line, separated by colons. At the same time, eliminate unnecessary spaces between keywords and variable names. Do NOT precede an ELSE statement or an abbreviated REM (') statement with a colon--BASIC automatically does this for you.

10. If you are unsure which sequence of commands to select for a particular task, experiment. Use our examples in this section to set up fair timing loops. The results will speak for themselves.

An unfortunate drawback to our suggestions is that you will wind up with completely unreadable programs. The solution is simple. Write the program with well placed subroutines and efficient algorithms. Comment the program with lots of REMs and leave plenty of blanks between variable names and keywords. Use the VARCOUNT, VARXREF, and LINEXREF utilities to help you determine the location and frequency of usage of all your variables and to help you define a series of proper DIM statements. Make sure you save a copy of the easy-to-read program. Then, after the program has been fully debugged and documented, use the BASMUNCH utility to strip out the comments and the extraneous spaces and compress the program into as little space as possible. Then renumber the program starting at line zero and incrementing by one's (we have not mentioned this trick before, but we know that you will find this a very easy way to produce marked increases in execution speed as well as memory savings -- we'll leave it to you to figure out why...) Save this as your execution copy. Don't lose the original program, though -- you will need to retain it in case you ever want to make changes. Remember, a compressed program is nearly impossible to change.

Section Three

BASIC SUBROUTINES

BASIC SUBROUTINES

A complete subroutine library can be a programmer's best friend, but for general-purpose subroutines to be useful, they must meet the following requirements:

1. They must be completely debugged. A programmer has enough on his mind when writing a program; he must be able to rely on the subroutines in his library.

2. By their very nature, general-purpose subroutines should not be too limited in their application.

3. Variables used by subroutines should not conflict with those already used by the main program.

We believe that the subroutines included here meet all of the above requirements. In addition, we have attempted to make them as efficient as possible in their use of both memory and processor time.

Variables Used

In order to make the subroutines as conflict-free as possible, we have reserved the variable names starting with the letter "O" for the exclusive use of the subroutines. Examples of these variables include O0, O9$, O4() and O1$(). All the variables that are not passed to the routines as parameters are initialized by the routines. These particular variables were chosen for use by the subroutines because of their infrequent use in BASIC programs. No doubt, one reason for this is the confusion between "O" and "0" that can easily arise. If you decide to use our philosophy in your own general-purpose subroutines, we suggest this procedure:

1. Write and debug the routine using meaningful variable names.

2. Ensure that any variables that do not have values assigned to them by the main program are initialized by the subroutine.

3. After the routine is debugged, change the variable names to those reserved for subroutine use.

Types of Routines Included

The routines included in this section are grouped into two categories. Section Three contains subroutines written in BASIC

which are easily incorporated into your own programs. Section Four is devoted to a collection of machine language routines that can greatly enhance the speed of your BASIC programs.

Line numbers 40000 - 49999 have been reserved for the BASIC routines, and 50000 - 59999 have been reserved for the machine language routines. All the routines begin with a REM line that contains the name of the routine in lowercase (reverse video) letters. The use of lowercase letters makes it easier to find comments in a screen listing.

Format of Routines

The first section of each routine contains a brief commentary on the use of the routine. If unusual programming techniques are used, note of this is made in this section.

Each routine also has a section detailing the variables used by the routine, the entry requirements and the exit conditions. Please be careful to note the effect the routine has on the variables: the values of the parameters, even though they are not intended as returned values, may be modified.

Finally, each routine shows a typical or sample call, as well as the actual listing of the subroutine. This is included to show a procedure for calling the routine; the actual use will, in most cases, be much more complex.

Using the routines with Cassette Systems

For those using cassette systems, three options are available. First, you could key in the program code from the listings supplied. (Due to the variables used and the opportunities they present for error, we do not recommend this procedure. Instead, we suggest that you purchase our tape, which contains all of the subroutines, along with a collection of valuable utility programs. We have included an order form at the end of the text.) The second method is to use a cassette merge system that simulates the operation of the MERGE statement of Disk BASIC. One such system is supplied with "Platinum Worksaver" (available from Platinum Software, P.O. Box 833, Plattsburgh, N.Y. 12901). A third alternative is to append the subroutines to your existing BASIC program as needed with the following method:

1. Make sure that the lowest line number in the program to be appended is higher than the highest line number of the program in memory.

2. CLOAD the first program.

3. Find the start addresses of the BASIC program by typing:

PRINT PEEK(25), PEEK(26) <ENTER>

Write down the two displayed values for use in step 6.

4. Now change the address of the start of BASIC to the end of the current program by typing the following:

POKE 25, PEEK(27) <ENTER>
POKE 26, PEEK(28) - 1 <ENTER>

5. Now CLOAD the second program. At this point if you LIST, RUN, or modify the program, you will only affect the second program. Your first program is still there, but it is invisible to BASIC.

6. To complete the appending of the two programs, type the following:

POKE 25, first value from step 3 <ENTER>
POKE 26, second value from step 3 <ENTER>

The two programs will now be appended. To append an additional program, just start over at step 3.

Transferring to Disk

If you are fortunate enough to have a disk system, we suggest that you transfer all the subroutines onto a special "subroutine disk". If you save them all in ASCII format, you can later use Disk BASIC's MERGE statement to add the subroutines to your programs as required.

GRNUMBER INIT (GOSUB 40000)
GRNUMBER MAIN (GOSUB 40020)

Purpose

This subroutine can be used to display a score, or any other number, on a high resolution graphics screen. The first part of the routine, "GRNUMBER INIT", puts the information required to draw the numbers into the array O1$. This routine must be called during program initialization. The second part, "GRNUMBER MAIN", draws the number at the top right corner of the screen. Any postive or negative number (less than 10 digits in length) can be displayed. If you wish to use a different screen location, modify the co-ordinates of the first DRAW statment in line 40020. To change the size of the digits displayed, change the "S4" in line 40020 to a different scale value -- "S8" will draw the numbers twice as large (use only multiples of 4, otherwise the digits will not be drawn properly).

The colors used by the DRAW command must match the foreground and background colors of your graphics screen. If the numbers do not print properly, change the values of "C" in O1$(10) in line 40010. The first "C" (currently "C0") must be set to the background color; the second (currently "C1") to the foreground. Note that you may have to reset the COLOR when returning from this routine to your own BASIC program.

Entry Requirements

1. Variable O1 must be set to the value of the number to be displayed.

2. GRNUMBER INIT must have been called before calling GRNUMBER MAIN.

Since the size of array O1$ is 11 elements, a DIM statement is not necessary. However, good programing practice calls for the DIM statement as in our sample call.

Exit Conditions

None.

Variables Used

O1, O2, O3, O4, O5, O1$, O1$().

Sample Call

```
00010  DIM O1$ (10)
00020  GOSUB 40000                          * DRAW data into array
00030  O1 = 1234                            * number to display
00040  PMODE 4                              * graphics mode
00050  SCREEN 1, 1

00060  GOSUB 40020                          * go display it
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listings

```
39999  REM grnumber init
40000  O1$(0) = "U12R6D12L6BL10"
      :O1$(1) = "BR6NU12BL16"
      :O1$(2) = "U6BU6R6D6NL6BD6L6BL10"
      :O1$(3) = "R6U6NL5U6L6BD12BL10"
      :O1$(4) = "BU6U6BR6D6NL6D6BL16"
      :O1$(5) = "BU6U6R6BD6NL6D6L6BL10"
40010  O1$(6) = "U12R6BD6NL6D6L6BL10"
      :O1$(7) = "BU12R6D12BL16"
      :O1$(8) = "U12R6D6NL6D6L6BL10"
      :O1$(9) = "BU6U6R6D6NL6D6L6BL10"
      :O1$(10) = "COU12R6D6NL6D6L6C1"
      :RETURN

40019  REM grnumber main
40020  DRAW "S4BM245,12;"
      :O1$ = STR$ (O5)
      :O2 = LEN (O1$) - 1
      :O1$ = RIGHT$ (O1$, O2)
      :FOR O3 = 0 TO O2 - 1
      :DRAW O1$(10) + O1$(VAL (MID$ (O1$, O2 - O3, 1)))
      :NEXT
      :DRAW O1$(10)
      :IF O5 < 0 THEN DRAW "BU6R6BD6BL16"
      :DRAW O1$(10)
40030  RETURN
```

BREAKDIS (GOSUB 40100)

Purpose

This subroutine disables the <BREAK> key during a BASIC program's execution, except during INPUT or LINE INPUT. Since BASIC scans for the <BREAK> after every instruction, an increase in speed will be achieved after this routine has been implemented. The routine modifies the RAM hook at &H019A, sending BASIC to a new routine at &H00F8. This new routine skips the check for the <BREAK> and <SHIFT><@> keys. TRON is also disabled by this routine. The new routine at &H00F8 is:

```
LEAS   2,S   remove return address
ANDCC  #$AF  enable interupts
JMP    $ADA5 re-enter interpreter
```

Before doing the required POKEs, the hook is checked to see if it has been modified. If it has, a RETURN is done; otherwise, the new routine is POKEd in, and a RUN is done. This RUN is necessary to properly initialize the routine. The JMP instruction at &H019A is first changed to an RTS while the new routine is being POKEd; it is then changed back to JMP.

Entry Requirements

None.

Exit Conditions

1. The <BREAK> key and trace function are disabled..

Variables Used

None.

Sample Call

```
00010  GOSUB 40100                    * call BREAKDIS
         .
         .                            * more instructions
         .
30000  END
```

Subroutine Listing

```
40099  REM breakdis
40100  IF PEEK (&H019B) = &H00 THEN RETURN
      :ELSE POKE &HF8,&H32
      :POKE &HF9,&H62
      :POKE &HFA,&H1C
      :POKE &HFB,&HAF
      :POKE &HFC,&H7E
      :POKE &HFD,&HAD
      :POKE &HFE,&HA5
      :POKE &H019A,&H39
      :POKE &H019B,&H00
      :POKE &H019C,&HF8
      :POKE &H019A,&H7E
      :RUN
```

BAUDRATE (GOSUB 40200)

Purpose

This subroutine is used to set the RS-232C port's baud rate delay value. It is very useful for people that own printers which operate at non-standard (other than 600 baud) rates. If the <ENTER> key is pressed in response to the prompt a default rate of 600 baud will be used. To select one of the other available rates, press the corresponding number key.

Entry Requirements

None.

Exit Conditions

1. The printer baud rate delay will be set to the one selected.

Variables Used

O1$.

Sample Call

```
00010  PRINT "PRINTER BAUD RATE INITIALIZATION"
00020  PRINT "PRESS 1 TO 6 TO SELECT NEW RATE"
00030  PRINT " OR <enter> FOR DEFAULT OF 600"
00040  PRINT
00050  GOSUB 40200                        * call BAUDRATE
       .
       .                                  * more instructions
       .
30000  END
```

Subroutine Listing

```
40199  REM baudrate
40200  PRINT " 1= 300  2= 600  3=1200  4=2400  5=4800  6=9600
         BAUD RATE?  ";
40210  O1$ = INKEY$
      :PRINT CHR$ (8);
      :IF O1$ = > "1" AND O1$ < = "6" THEN 40230
      :ELSE IF O1$ = CHR$ (13) THEN O1$ = "2"
      :GOTO 40230
40220  PRINT CHR$ (128);
      :GOTO 40210
40230  POKE 150, VAL (MID$ ("180 87 41 18  7  1",(VAL (O1$) - 1
       ) * 3 + 1,3))
      :PRINT O1$
      :RETURN
```

JOYSTICK (GOSUB 40300)

Purpose

When using joysticks, the right horizontal joystick value, JOYSTK(0), must always be read before any other value can be read. This is due to the fact that the routine in Color BASIC that reads the joystick ports is only executed on a JOYSTK(0). When JOYSTK(1) to JOYSTK(3) are used, the values stored at memory locations $015A to $015D are assigned to the variable without actually reading the ports. This means that if you are using the left joystick in a game, you must read the right horizontal joystick, even though you have no use for the value.

This short routine forces Color BASIC to update the values, and then assigns them to O0 and O1. The trigger status is also checked and returned in O2.

For the proper operation of this routine the variable OJ must be set. For one method of setting this value see our subroutine JOYORKEY.

Entry Requirements

1. OJ must be set as follows:

   to read the right joystick OJ = &HO15A,
   to read the left joystick OJ = &HO15C.

Exit Conditions

1. O0 - horizontal data.
2. O1 - vertical data.
3. O2 - trigger status with the following meanings:

   0 = both triggers pressed,
   1 = left trigger pressed,
   2 = right trigger pressed,
   3 = no triggers pressed.

Variables Used

OJ, O0, O1, O2.

Sample Call

```
00010  OJ = &H015A                          * right joystick
      :GOSUB 40300                          * go read
00020  PRINT "HORIZONTAL = "O0               * display results
00030  PRINT "VERTICAL   = "O1
00040  PRINT "TIGGER STATUS = ";
00050  IF O2 = 0 THEN PRINT "BOTH";
00060  IF O2 = 1 THEN PRINT "LEFT";
00070  IF O2 = 2 THEN PRINT "RIGHT";
00080  IF O2 = 3 THEN PRINT "NONE";
00090  PRINT " PRESSED"
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
40299  REM joystick
40300  EXEC &HA9DE
      :O0 = PEEK (OJ + 1)
      :O1 = PEEK (OJ)
      :O2 = (PEEK (&HFF00) AND 3)
      :RETURN
```

JOYORKEY (GOSUB 40400)

Purpose

Games and other programs using joysticks for input should give the user the option of deciding which joystick he would like to use or, if no joysticks are available, the option of using the keyboard. This subroutine will display the message "* * * PRESS THE JOYSTICK BUTTON OR <K> TO START " at the bottom of the text screen. Since this message is longer than 32 characters, we have developed a horizontal scroll routine that displays the message 30 characters at a time.

The message continues to scroll until one of the joystick triggers is pressed or the <K> key is pressed.

This routine is designed to be used in conjuction with JOYSTICK which requires the variable OJ set to the left or right joystick.

Entry Requirements

None.

Exit Conditions

1. OJ = 0 if the keyboard is selected,
   = &H015A if the right joystick is selected,
   = &H015C if the left joystick is selected.

Variables Used

OO$, OO, OJ.

Sample Call

```
00010  GOSUB 40400                          * call JOYORKEY
00020  PRINT "USER WANTS TO USE THE ";  * display results
00030  IF OJ = 0 THEN PRINT "KEYBOARD"
00040  IF OJ = &H015A THEN PRINT "RIGHT JOYSTICK"
00050  IF OJ = &H015C THEN PRINT "LEFT JOYSTICK"
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
40399  REM joyorkey
40400  OO$ = "* * * PRESS THE JOYSTICK BUTTON OR <K> TO START "
40410  FOR OO = 30 TO 1 STEP - 1
      :PRINT @480 + OO, MID$ (OO$,OO,1);
      :NEXT
      :OO$ = MID$ (OO$,2) + LEFT$ (OO$,1)
      :IF INKEY$ = "K" THEN OJ = 0
      :RETURN
      :ELSE IF (PEEK (&HFFOO) AND 3) = 1 THEN OJ = &H15A
      :RETURN
      :ELSE IF (PEEK (&HFFOO) AND 3) = 2 THEN OJ = &H15C
      :RETURN
      :ELSE 40410
```

TOBASIC (GOSUB 40500)

Purpose

This subroutine should be included in all of your BASIC programs. When the <Y> key is pressed in response to the "RETURN TO BASIC?" prompt, the routine first CLOSEs any files that are still OPEN. If disks are present their drive heads are reset to track zero (this will prevent the annoying "thunking" noise that the drives make from time to time when searching for the directory track immediately after power-up). A POKE is done to reset the computer to memory map type 0, and BASIC's cold start routine is entered. A new program can now be loaded, without having to worry about ML routines, CLEARs and PCLEARs "left over" from your last program. If the <N> key is pressed, a RETURN is done; any other keypress is ignored.

A word of warning: Since the program in memory is effectively destroyed, don't append this routine until the program is completely debugged. Nothing is more infuriating than to program for hours, try the program, and exit via this routine before the latest version has been saved.

Entry Requirements

None.

Exit Conditions

1. Control is passed back to BASIC after files have been CLOSEd, drive heads restored to track zero and a cold start has been executed.

Variables Used

O1, O2, O1$.

Sample Call

```
00010  GOSUB 40500                          * call TOBASIC
00020  PRINT "TOBASIC WAS CALLED BUT NOT EXECUTED"
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
40499  REM tobasic
40500  PRINT "ARE YOU SURE YOU WISH TO RETURN TO BASIC
        (Y OR N)?  ";
40510  PRINT CHR$ (8);
      :O1$ = INKEY$
      :IF O1$ = "N" THEN RETURN
      :ELSE IF O1$ < > "Y" THEN PRINT CHR$ (128);
      :GOTO 40510
      :ELSE CLOSE
      :IF PEEK (&HC000) < > 126 THEN O1 = PEEK (&HC006) * 256 +
       PEEK (&HC007)
      :FOR O2 = 0 TO 3
      :POKE O1,0
      :POKE O1 + 1,O2
      :EXEC PEEK (&HC004) * 256 + PEEK (&HC005)
      :NEXT
40520  POKE &H71,0
      :POKE &HFFDE,0
      :EXEC &HA027
```

DPEEK (GOSUB 40600)

Purpose

This short "double peek" subroutine will return a value representing the contents of two successive memory locations. This is very useful when you are determining addresses or values for machine language programs. If the address specifed is out of range, a RETURN is done before the PEEK.

Entry Requirements

1. O1 represents the address of the first of the two memory locations being PEEKed.

Exit Conditions

1. O2 contains the value of memory locations O1 and O1 + 1.

Variables Used

O1, O2.

Sample Call

```
00010  O1 = &H1234                          * address to examine
00020  GOSUB 40600                          * do double PEEK
00030  PRINT "THE TWO BYTE VALUE AT $";     * display results
00040  PRINT HEX$ (O1);
00050  PRINT " = $"; HEX$ (O2)
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
40599  REM dpeek
40600  IF O1 < 0 OR O1 > 65534 THEN RETURN
      :ELSE O2 = PEEK (O1) * 256 + PEEK (O1 + 1)
      :RETURN
```

DPOKE (GOSUB 40700)

Purpose

This "double poke" routine, meant to be used in conjunction with DPEEK, POKEs the two byte value contained in O2 into memory locations O1 and O1 + 1.

Entry Requirements

1. O1 - the memory address at which the two byte data is to be stored.

2. O2 - the 2 byte value to be stored at O1.

Exit Conditions

None.

Variables Used

O1, O2.

Sample Call

```
00010  O1 = &H0034                          * address
00020  O2 = &H1234                          * value
00030  GOSUB 40700                          * go double poke
00040  PRINT HEX$ (O1) " = $" HEX$ (PEEK (O1))
00050  PRINT HEX$ (O1 + 1) " = $" HEX$ (PEEK (O1 + 1))
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
40699  REM dpoke
40700  IF O1 < 0 OR O1 > 65534 OR O2 < 0 OR O2 > 65535 THEN
       RETURN
       :ELSE POKE O1, INT (O2 / 256)
       :POKE O1 + 1, O2 - INT (O2 / 256) * 256
       :RETURN
```

SYSTEM (GOSUB 40800)

Purpose

With the profusion of ROMs for the Color Computer and their individual peculiarities, it is often desirable to know which versions of BASIC are present. This subroutine will let the program determine the version numbers for the ROMs present, as well as the memory size of the computer (16K or 32K).

This routine presumes that EXTENDED COLOR BASIC is present.

Entry Requirements

None

Exit Conditions

1. O1$ is a 4-byte string that contains the following information:

```
Byte 1 - memory size (16K or 32K)
     2 - BASIC version number
     3 - EXTENDED BASIC version number
     4 - DISK BASIC (if present) version number
```

Variables Used

O1, O1$.

Sample Call

```
00010  GOSUB 40800                          * go check system
00020  PRINT "SYSTEM CONFIGURATION"         * display results
00030  PRINT
00040  PRINT "MEMORY SIZE = " ASC (MID$ (O1$,1)) "K"
00050  PRINT "COLOR BASIC V 1." MID$ (O1$,2,1)
00060  PRINT "EXTENDED BASIC V 1." MID$ (O1$,3,1)
00070  IF RIGHT$ (O1$,1) <> CHR$ (0) THEN PRINT "DISK BASIC
        V 1." RIGHT$ (O1,1)
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
40800  O1$ = STRING$ (4,0)
       :O1 = PEEK (116)
       :O1 = (O1 = 127) * - 32 + (O1 = 63) * - 16
       :MID$ (O1$,1) = CHR$ (O1)
       :MID$ (O1$,2) = CHR$ (PEEK (&HA155))
       :MID$ (O1$,3) = CHR$ (PEEK (&H80FF))
       :O1 = PEEK (&HC005)
       :IF O1 = &H6C THEN  O1 = 48
       :ELSE IF O1 = &H5F THEN O1 = 49
       :ELSE O1 = 0
40810  MID$ (O1$,4) = CHR$ (O1)
       :RETURN
```

GETDATE (GOSUB 40900)

Purpose

GETDATE is an input routine that prompts a user to type in a date in the "DD/MM/YY" format. The non-destructive cursor can be moved to the left and right with the arrow keys. Only the numbers "0" to "9" are allowed to be input. When the <ENTER> key is pressed, the routine checks the date entered to be sure it is a valid date. If it is not valid, an error message is shown and, after a brief delay, the cursor reappears.

The error checking is done in the following manner. First, the month is checked. If it is not between "01" and "12", it is not valid and the routine goes to the error handler. Next, the year is determined. If it is evenly divisible by four, it is considered to be a leap year. This is required for the next step which checks the day. If the day is "00", or if it is greater than the number of days in the month specified by the previous step, the error routine is called. If everything is O.K. at this point the routine returns to the main program. The day, month and year values are returned so your main program can do further checking. This may include checking to make sure the date is later than a preset "prior date".

A minor point, in case you decide to use this routine for some other purpose, is that the leap year calculataion is only valid for the years 1901 to 2099. The rule for determining leap year is:

> A leap year is any year evenly divisible by 4, unless it is also evenly divisible by 100, in which case it must be evenly divisible by 400.

For this reason the routine works for the year 2000, which is a leap year; but the year 2100, which is not a leap year, would be incorrectly accepted by the subroutine.

Entry Requirements

None.

Exit Conditions

1. O8$ - contains the date in the format dd/mm/yy.
2. O3 - numeric value of the month.

3. O4 - numeric value of the day.

4. O5 - numeric value of the year.

Variables Used

O1, O2, O3, O4, O5, O1$, O8$.

Sample Call

```
00010  DIM M$(12)                          * for month-names
00020  FOR T = 1 TO 12                     * fill the array
      :READ M$(T)
      :NEXT
      :DATA JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
       AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
00010  GOSUB 40900                         * go get a date
00030  PRINT "THE DATE IS ";
00040  PRINT M$(O3); O4; "19"; O5
      .
      .                                    * more instructions
      .
30000  END
```

Subroutine Listing

```
40899  REM getdate
40900  PRINT
      :PRINT " DATE AS dd/mm/yy: ";
      :O1 = PEEK (136) * 256 + PEEK (137) - 1024
      :O8$ ="00/00/00"
      :O2 = 0
40910  PRINT @O1,O8$ " ";
      :O1$ = INKEY$
      :IF O1$ = CHR$ (13) THEN 40940
40920  PRINT @ O1 + O2, CHR$ (128);
      :IF O1$ = CHR$ (8) AND O2 > 0 THEN O2 = O2 - 1 + (O2 = 3)
       + (O2 = 6)
      :ELSE IF O1$ = CHR$ (9) AND O2 < 8 THEN O2 = O2 + 1 - (O2
       = 1) - (O2 = 4)
      :ELSE IF O1$ > CHR$ (47) AND O1$ < CHR$ (58) AND O2 < 8
       THEN MID$ (O8$,O2 + 1,1) = O1$
      :O2 = O2 + 1 - (O2 = 1) - (O2 = 4)
40930  GOTO 40910
```

```
40940  O3 = VAL (MID$ (O8$,4,2))
      :IF O3 > 12 OR O3 = 0 THEN 40950
      :ELSE O4 = VAL (O8$)
      :O5 = VAL (MID$ (O8$,7))
      :IF O4 < 1 OR O4 > VAL (MID$ ( "30" + MID$ (STR$ (28 -
       (O5 / 4 = INT (O5 / 4))),2) + "3130313031313031303131",
       (O3 - 1) * 2 + 1,2)) THEN 40950
      :ELSE RETURN
40950  PRINT @O1, "Invalid date";
      :SOUND 100,1
      :FOR O3 = 1 TO 500
      :NEXT
      :PRINT @O1, STRING$ (12,32);
      :GOTO 40910
```

CASSNAME (GOSUB 41000)

Purpose

This input subroutine prompts the user to input an eight-character cassette filename. The non-destructive cursor can be moved left and right with the arrow keys. The input is ended when the <ENTER> key is pressed. Any name input via this routine will be a legal cassette file or program name.

Entry Requirements

None.

Exit Conditions

1. O9$ is an 8 character string containing the filename.

Variables Used

O1, O2, O1$, O9$.

Sample Call

```
00010 GOSUB 41000                    * get a casette name
00020 OPEN"I", #-1, O9$              * do something with it
      .
      .                              * more instructions
      .
30000  END
```

Subroutine Listing

```
40999  REM cassname
41000  PRINT
      :PRINT "CASSETTE FILENAME:" CHR$ (175);
      :O1 = PEEK (136) * 256 + PEEK (137) - 1024
      :O9$ = STRING$ (8,32)
      :O2 = 0
41010  PRINT @O1,O9$ CHR$ (175);
      :O1$ = INKEY$
      :IF O1$ = CHR$ (13) THEN PRINT
      :RETURN
      :ELSE PRINT @O1 + O2, CHR$ (128);
      :IF O1$ = CHR$ (8) AND O2 > 0 THEN O2 = O2 - 1
      :ELSE IF O1$ = CHR$ (9) AND O2 < 7 THEN O2 = O2 + 1
      :ELSE IF O1$ > CHR$ (31) AND O2 < 8 THEN MID$ (O9$,
       O2 + 1,1) = O1$
      :O2 = O2 + 1
41020  GOTO 41010
```

INKEY$ (GOSUB 41100)

Purpose

This general-purpose subroutine pauses program execution until a key is pressed. If the key is a lowercase letter it is converted to uppercase. The routine then returns to the main program.

Entry Requirements

None.

Exit Conditions

1. O1$ is a one letter string containing the key pressed. If the key was a lowercase letter, it will have been changed to the equivalent uppercase letter.

Variables Used

O1, O1$.

Sample Call

```
00010  GOSUB 41100                    * wait for a keypress
00020  PRINT O1$ " WAS PRESSED"       * display
results
       .
       .                              * more instructions
       .
30000  END
```

Subroutine Listing

```
41099  REM inkey$
41100  O1$ = INKEY$
      :IF O1$ = "" THEN 41100
      :ELSE IF O1$ > "Z" THEN O1$ = CHR$ (ASC (O1$) AND 95)
41110  RETURN
```

KEYINPT (GOSUB 41200)

Purpose

This subroutine is very useful when a specific single key input is desired. The string passed to the routine contains the allowable inputs. The routine will flash a cursor until an acceptable key is pressed, at which time the character will be displayed at the cursor position. Program execution is now passed back to the main program. The position of the character input in the allowable character string is also returned for use in an ON GOTO selection.

Entry Requirements

1. O1$ must contain a list of acceptable characters to be input.

Exit Conditions

1. O2$ contains the key pressed. If it was a lowercase letter, it will have been changed to uppercase.

2. O1 - the postion of O2$ in O1$.

Variables Used

O1, O1$, O2$.

Sample Call

```
00010  O1$="123"                          * allowable inputs
00020  GOSUB 41200                        * wait for a keypress
00030  ON O1 GOTO 100, 200, 300           * go to right line
00100  PRINT "1 PRESSED"
      :GOTO 20
00200  PRINT "2 PRESSED"
      :GOTO 20
00300  PRINT "3 PRESSED"
      :GOTO 20
       .
       .                                  * more instructions
       .
30000  END
```

Subroutine Listing

```
41199  REM keyinpt
41200  O2$ = INKEY$
      :IF O2$ = "" THEN O1 = O1 * (O1 > 0) * (O1 < 16) + 1
      :POKE PEEK (&H88) * 256 + PEEK (&H89), - 128 * (O1 > 7)
       - 143 * (O1 < 8)
      :GOTO 41200
      :ELSE O2$ = CHR$ (ASC (O2$) + 32 * (O2$ > "Z"))
      :O1 = INSTR (O1$,O2$)
      :IF O1 THEN PRINT O2$;
      :RETURN
      :ELSE 41200
```

LINEINPT (GOSUB 41300)

Purpose

This subroutine allows keyboard input of a string of a specifed maximum length at the specified cursor position. A solid black cursor is shown. The <LEFT ARROW> key can be used to backspace one position. Pressing <SHIFT><LEFT ARROW> erases the entire line and returns the cursor to the beginning. Input is not allowed past the specified maximum length. Input is terminated when the <ENTER> key is pressed.

Entry Requirements

1. O1 - initial cursor position.

2. O2 - maximum allowable length.

Exit Conditions

1. O7$ - contains the input data.

Variables Used

O1, O2, O3, O1$, O7$.

Sample Call

```
00010  O1 = 100                          * length
00020  O2 = 64                           * position
00030  GOSUB 41200                       * get some input
       .
       .                                 * more instructions
       .
30000  END
```

Subroutine Listing

```
41299  REM lineinpt
41300  O7$ = STRING$ (O2,32)
      :O3 = 1
41310  PRINT @O1,O7$ CHR$ (134)
41320  PRINT @O3 + O1 - 1, CHR$ (128);
      :O1$ = INKEY$
      :IF O1$ = "" THEN 41320
      :ELSE IF O1$ > CHR$ (31) AND O3 < = O2 THEN MID$ (O7$,
       O3) = O1$
      :PRINT @O3 + O1 - 1,O1$;
      :O3 = O3 + 1
      :GOTO 41320
41330  IF O1$ = CHR$ (13) THEN O7$ = LEFT$ (O7$,O3 - 1)
      :RETURN
      :ELSE IF O1$ = CHR$ (21) THEN 41300
      :ELSE IF O1$ = CHR$ (8) AND O3 > 1 AND O3 < = O2 THEN MID$
       (O7$,O3) = " "
      :O3 = O3 - 1
      :ELSE IF O1$ = CHR$ (8) AND O3 > 1 THEN O3 = O3 - 1
41340  GOTO 41310
```

READY# (GOSUB 41400)

Purpose

This subroutine will display a message to ready the printer, cassette player or disk drive 0 to 3. The word ENTER in the prompt will flash until the <ENTER> key is pressed. No check of the device is made in the routine -- that is the responsibility of your main program.

Entry Requirements

1. O1 - must contain the device number to ready:

   -2 for the printer
   -1 for the cassette
    0 to 3 for the disk drives.

(The prompt message is displayed at the current cursor position.)

Exit Conditions

None.

Variables Used

O1, O2, O3.

Sample Call

```
00010  O1 = -2                          * ready printer
00020  GOSUB 41400                      * call READY#
       .
       .                                * more instructions
       .
30000  END
```

Subroutine Listing

```
41399  REM ready#
41400  O2 = PEEK (136) * 256 + PEEK (137) - 992
      :O2 = O2 + (O2 = 512) * 32
      :PRINT @O2 - 28,;
      :IF O1 = - 2 THEN PRINT "    PREPARE  PRINTER    ";
      :ELSE IF O1 = - 1 THEN PRINT "    PREPARE CASSETTE
         ";
      :ELSE PRINT " PREPARE DISK DRIVE #"O1;
41410  PRINT @O2 + 4, " PRESS enter WHEN READY ";
      :FOR O3 = 1 TO 70
      :IF INKEY$ = CHR$ (13) THEN 41420
      :ELSE NEXT
      :PRINT @O2 + 4, " PRESS ENTER WHEN READY ";
      :FOR O3 = 1 TO 70
      :IF INKEY$ = CHR$ (13) THEN 41420
      :ELSE NEXT
      :GOTO 41410
41420  O3 = 100
      :NEXT
      :RETURN
```

PRESCON1 (GOSUB 41500)

Purpose

This subroutine will display a flashing "PRESS ANY KEY TO CONTINUE" message across the bottom of the text screen. The message will continue to flash until a key is pressed.

Entry Requirements

None.

Exit Conditions

None.

Variables Used

O1$, O2$.

Sample Call

```
00010  GOSUB 41500                          * call PRESCON1
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
41499  REM prescon1
41500  O1$ = INKEY$
      :POKE 1535, 128
      :O2$ = CHR$ (128)
      :PRINT @480, STRING$ (31,128);
      :FOR O1 = 1 TO 99
      :IF INKEY$ < > "" THEN 41510
      :ELSE NEXT
      :PRINT @483, "press"O2$ "any"O2$ "key"O2$ "to"O2$ "co
       ntinue";
      :FOR O1 = 1 TO 99
      :IF INKEY$ < > "" THEN 41510
      :ELSE NEXT
      :GOTO 41500
41510  O1 = 100
      :NEXT
      :RETURN
```

PRESCON2 (GOSUB 41600)

Purpose

This subroutine, similiar to PRESCON1, allows the entire screen to be used for text display. When the routine is called, a standard cursor will be displayed at the bottom right corner of the screen. Program execution will then pause until any key is pressed.

Entry Requirements

None.

Exit Conditions

1. The text screen is cleared before the routine returns control to the main program.

Variables Used

None.

Sample Call

```
00010  GOSUB 41500                    * call PRESCON2
       .
       .                              * more instructions
       .
30000  END
```

Subroutine Listing

```
41599  REM prescon2
41600  POKE &H88,&H05
      :POKE &H89,&HFF
      :EXEC &HA1B1
      :CLS
      :RETURN
```

PRINTON (GOSUB 41700)

Purpose

This subroutine checks to see if the printer is ready. If it is not, the flashing message "THE PRINTER IS NOT ON" will be displayed. This message will continue to be displayed until either one of the following conditions are met: First, the printer can be turned on. Second, the <ENTER> key can be pressed. If exit occurs because the user pressed <ENTER>, O3 will contain a value other than zero that can be used by the main program to abort the print routine.

Entry Requirements

None.

Exit Conditions

1. O1 - equals zero if the printer was ready.

Variables Used

O1, O2, O3, O1$, O2$.

Sample Call

```
00010  GOSUB 41700                    * check printer
00020  IF O1 THEN GOTO 1000           * main menu?
00030  PRINT#-2,"DATA"                * something to print
       .
       .                              * more instructions
       .
30000  END
```

Subroutine Listing

```
41699  REM printon
41700  O1 = PEEK (&H88) * 256 + PEEK (&H89) - 1019
      :O1$ = CHR$ (128)
      :O2$ = INKEY$
41710  O3 = (PEEK (&HFF22) AND 1)
      :IF O3 = 0 OR INKEY$ = CHR$ (13) THEN RETURN
      :ELSE PRINT @O1, "the"O1$ "printer"O1$ "is" O1$ "not"O1
       $ "on";
      :FOR O2 = 1 TO 300
      :NEXT
      :PRINT @O1, STRING$ (21,32);
      :FOR O2 = 1 TO 200
      :NEXT
      :GOTO 41710
```

NEATPRNT (GOSUB 41810)

Purpose

Formatting text for display on the text screen is always a time-consuming, tedious chore. The NEATPRNT subroutine will format the text sent to it and display it without any word wrap (words being split at the edge of the screen).

Entry Requirements

1. O1$ - contains the text to be displayed.
2. The display commences at the current cursor position.

Exit Conditions

None.

Variables Used

O1, O2, O1$.

Sample Call

```
00010  O1$ = "THIS IS SAMPLE TEXT TO BE DISPLAYED ON THE TEXT
       SCREEN."
00020  GOSUB 41800                          * go display it
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
41799  REM neatprnt
41800  O1 = 1
41810  O2 = INSTR (O1,O1$, " ") - O1
      :IF O2 = 0 THEN O2 = LEN (O1$)
      :ELSE IF O2 < 0 THEN O2 = LEN (O1$)
41820 IF POS (0) + O2 > 31 THEN PRINT
41830 PRINT MID$ (O1$,O1,O2) " ";
      :IF O2 + O1 = > LEN (O1$) THEN RETURN
      :ELSE O1 = O1 + O2 + 1
      :GOTO 41810
```

SCREENPT (GOSUB 41900)

Purpose

This subroutine prints the contents of the text screen on the printer. Any graphics characters on the screen are replaced by asterisks.

Entry Requirements

1. Since the routine will cause the computer to lock-up if the printer is not ready, PRINTON should be used before this routine is called.

Exit Conditions

None.

Variables Used

O1, O2, O3.

Sample Call

```
00010  GOSUB 41900                          * call SCREENPT
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
41899  REM screenpt
41900  PRINT # - 2
      :PRINT # - 2, TAB (24);
      :FOR O1 = 0 TO 15
      :FOR O2 = 0 TO 31
      :O3 = PEEK (((O1 * 32) + O2) + 1024)
      :O3 = O3 + (O3 > 95 AND O3 < 128) * 64 - (O3 > = 0 AND O3 <
       32) * 96
      :IF O3 > 128 THEN O3 = 42
41910  PRINT # - 2, CHR$ (O3);
      :NEXT
      :PRINT # - 2
      :PRINT # - 2, TAB (24);
      :NEXT
      :PRINT # - 2
      :RETURN
```

MENUDISP (GOSUB 42000)

Purpose

Neatly formatted menus are time-consuming to create, but they do make a program look more professional. MENUDISP displays the desired menu information neatly centered, complete with a header and a "selection" message at the bottom of the screen.

This subroutine does not clear the screen. Instead, it uses PRINT @ statements to position the text on screen -- over top of the existing display. This feature makes MENUDISP a natural subroutine to use with the background color flasher routine described in the next section.

Entry Requirements

1. O2$ - must contain the text to be displayed. Lines must be separated from each other by "@" symbols. If any line is longer than 32 characters in length, the routine will not format the display properly.

2. O3$ - may contain the header and footer messages seperated by "@"s. If either message is longer than 32 characters, an improper display will result. If O3$ is undefined or does not contain an "@" no header or footer will be displayed. If you wish to display only a header, just end the definition with an "@", if you wish a footer only, begin the definition with an "@".

3. The screen must be cleared to the desired background pattern before the routine is called.

Exit Conditions

1. O2$ may be modified.

Variables Used

O1, O2, O3, O4, O5, O6, O2$, O3$.

Sample Call

```
00010  PRINT@0, STRING$ (160, 191); STRING$ (192, 159); STRING$
       (159, 191);                          * background display
00020  POKE 1535, 191
00030  O2$ = "1. ADD DATA@2. INPUT DATA FROM CASSETTE@3. SAVE
       DATA TO CASSETTE@4. END PROGRAM" * menu data
00040  O3$ = " MAIN MENU @ SELECTION? "
```

```
00050  GOSUB 42000                          * display the menu
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
41999  REM menudisp
42000  O6 = INSTR (O3$, "@")
      :O6 = O6 + (O6 > 0)
      :O2$ = O2$ + "@"
      :O1 = 1
      :O4 = 0
      :O5 = 0
      :FOR O2 = 1 TO LEN (O2$)
      :O2 = INSTR (O1,O2$, "@")
      :O3 = O2 - O1
      :O4 = - O3 * (O3 > O4) - O4 * (O3 < = O4)
      :O1 = O2 + 1
      :O5 = O5 + 1
      :NEXT
      :PRINT @ - 32 * (O5 < 12) + 16 - O6 / 2, LEFT$ (O3$,O6);
42010  O3 = (8 - INT (O5 / 2)) * 32 + 16 - O4 / 2
      :O1 = 1
      :FOR O2 = 1 TO LEN (O2$)
      :O2 = INSTR (O1,O2$, "@")
      :PRINT @O3, MID$ (O2$,O1,O2 - O1) STRING$(O4 -
       (O2 - O1), 32);
      :O3 = O3 + 32
      :O1 = O2 + 1
      :NEXT
      :PRINT @496 + 32 * (O5 < 11) - (LEN (O3$) - O6 - 2) / 2,
       MID$ (O3$,O6 + 2);
      :RETURN
```

CHKDRIVE (GOSUB 42100)

Purpose

This subroutine checks the specified drive to see if it has a disk ready to be read or written to. If the drive is ready, O1 will be equal to zero on return.

This routine uses the DSKCON ROM routine documented in the Disk System Owners Manual to read track 17, sector 2 (the file allocation table). If the read is successful the drive is presumed to be ready.

Entry Requirements

1. O9 must be set to the number of the drive to be checked.

Exit Conditions

1. O1 will be zero if the drive is ready. Any other value represents a DSKCON error code (see your Disk System Owner's Manual).

Variables Used

O1, O9.

Sample Call

```
00010  O9 = 0                          * drive number
00020  GOSUB 42100                     * see if it's ready
00030  IF O1 THEN GOTO 100             * error message?
       .
       .                               * more instructions
       .
30000  END
```

Subroutine Listing

```
42099  REM chkdrive
42100  O1 = PEEK (&HC006) * 256 + PEEK (&HC007)
      :POKE O1,2
      :POKE O1 + 1,09
      :POKE O1 + 2,17
      :POKE O1 + 3,2
      :POKE O1 + 4,&H06
      :POKE O1 + 5,&H00
      :EXEC PEEK (&HC004) * 256 + PEEK (&HC005)
      :O1 = PEEK (O1 + 6)
      :RETURN
```

DIR (GOSUB 42200)

Purpose

Even though the DIR command can be used in a BASIC program, its unattractive display which scrolls off the screen should be avoided. This directory subroutine fills the void by displaying the directory of the specified drive in a neat two column display. The file types and sizes are not displayed. The display pauses until a key is pressed each time the screen is filled. When all the files have been listed, the number of free granules is displayed. The routine will again wait for a keypress before returning to the main program.

The operation of this routine is nearly as fast as that of the DIR statement. This is accomplished by using the DSKCON ROM routine to read the directory sectors directly into O1$. In order to do this the string descriptor of O1$ is first set to the address of the disk buffer used by DKSCON. Now, when DSKCON is called the sector data will be stored in O1$, which can now be manipulated by the rest of the routine which determines and prints the file names.

Entry Requirements

1. O9 must contain the number of the drive to be read. Note that drive status is not checked by this routine. For this reason, CHKDRIVE should be called first.

Exit Conditions

1. O6 equals the number of free granules remaining on the disk.

Variables Used

O1, O2, O3, O4, O5, O6, O9, O1$.

Sample Call

```
00010  O9 = 0                          * drive number
00020  GOSUB 42200                     * go display directory
       .
       .                               * more instructions
       .
30000  END
```

Subroutine Listing

```
42199  REM dir
42200  O1$ = ""
      :O1 = VARPTR (O1$)
      :POKE O1,255
      :POKE O1 + 2,&H06
      :POKE O1 + 3,0
      :O1 = PEEK (&HC004) * 256 + PEEK (&HC005)
      :O2 = PEEK (&HC006) * 256 + PEEK (&HC007) + 3
42210  O6 = FREE (O9)
      :CLS
      :O3 = 0
      :FOR O4 = 3 TO 11
      :POKE O2,O4
      :EXEC O1
      :FOR O5 = 1 TO 255 STEP 32
      :IF MID$ (O1$,O5,1) = CHR$ (0) THEN 42230
      :ELSE IF MID$ (O1$,O5,1) = CHR$ (255) THEN O4 = 11
      :GOTO 42230
      :ELSE O3 = O3 + 1
      :PRINT "  " MID$ (O1$,O5,8) "." MID$ (O1$,O5 + 8,
       3);
      :IF O3 < 32 THEN PRINT "  ";
42220  IF O3 > 31 THEN PRINT " ";
      :EXEC &HA1B1
      :CLS
      :O3 = 0
42230  NEXT
      :NEXT
      :IF O3 = 31 THEN EXEC &HA1B1
      :CLS
      :ELSE IF POS (0) < > 0 THEN PRINT
42240 PRINT TAB (6)O6 "FREE GRANS ";
      :EXEC &HA1B1
      :RETURN
```

DISKNAME (GOSUB 42300)

Purpose

This routine prompts the user to input a disk filename. The prompt consists of the text "DISK FILENAME:" and a defined area for the non-destructive cursor. The fields for the name, extension and drive number are also indicated. The default extension "DAT" and drive number "0" are already present in the filename, but these values may be changed as required by the user.

The cursor can be moved left and right with the <LEFT ARROW> and <RIGHT ARROW> keys. It can be moved to the next field position with the <DOWN ARROW> key. If this key is pressed when the cursor is at the space reserved for the drive number, it will return to the first position of the filename.

The routine prevents the entry of a period (.) slash (/) or colon (:) during the input of the name and extension. Only the numbers 0, 1, 2 and 3 are allowed when inputting the drive number.

If your program uses different default drive numbers or filename extensions, edit the O9$=" .DAT:0" in line 42300 to reflect your program's needs. You can also limit the allowable drive numbers by modifying O1$ in the same line.

Entry Requirements

None.

Exit Conditions

1. O9$ is a 14 character string containing the filename. Its format is "FILENAME.EXT:D".

Variables Used

O1, O2, O1$, O2$, O9$.

Sample Call

```
00010  GOSUB 42300                    * get data filename
00020  OPEN "I", 1, O9$               * do something with it
       .
       .                              * more instructions
       .
30000  END
```

Subroutine Listing

```
42299  REM diskname
42300  PRINT
      :PRINT "DISK FILENAME:" CHR$ (175);
      :O1 = PEEK (136) * 256 + PEEK (137) - 1024
      :O9$ = "         .DAT:0"
      :O2 = 0
      :O2$ = "0123"
42310  PRINT @O1,O9$ CHR$ (175);
      :O1$ = INKEY$
      :IF O1$ = CHR$ (13) THEN PRINT
      :RETURN
      :ELSE PRINT @O1 + O2, CHR$ (128);
      :IF O1$ = CHR$ (8) AND O2 > 0 THEN O2 = O2 - 1 +
       (O2 = 9) + (O2 = 13)
      :ELSE IF O1$ = CHR$ (9) AND O2 < 13 THEN O2 = O2 + 1 -
       (O2 = 7) - (O2 = 11)
42320  IF INSTR (".:/",O1$) THEN O1$ = ""
      :ELSE IF O1$ = CHR$ (10) THEN O2 = (O2 < 8) * - 9 +
       (O2 < 12 AND O2 > 8) * - 13
      :ELSE IF O2 = 13 AND INSTR (O2$, O1$) = 0 THEN O1$ = ""
      :ELSE IF O1$ > CHR$ (31) AND O2 < 14 THEN MID$ (O9$,
       O2 + 1, 1) = O1$
      :O2 = O2 + 1 - (O2 = 7) - (O2 = 11)
42330  GOTO 42310
```

FILEXIST (GOSUB 42400)

Purpose

This subroutine checks the disk in the specified drive to determine if the specified filename exists. This routine can be useful if a new file is to be created, as well as when opening an existing one.

Note that the filename returned by DISKNAME is completely compatible with this routine.

This routine uses an undocumented ROM routine contained in Disk BASIC to check the directory. This routine is located at $C657 in version 1.0 and $C68C in version 1.1. It may be moved to another location in future releases, in which case the routine will need to be modified.

Entry Requirements

1. O9$ must contain the disk name in the following format:

   FILENAME/EXT:O

   Either a period (.) or slash (/) may be used as a delimiter between the name and extension. Note that the defaults are drive 0 and /DAT.

Exit Conditions

1. O9$ is not affected.

2. O5 = -1 if filename is illegal,
   = 0 if the file does not exist,
   > 0 indicates that the file is present.

Variables Used

O1, O5, O2$, O3$, O9$.

Sample Call

```
00010  O9$ = "LETTER/DAT:O"                  * target filename
00020  GOSUB 42400                           * see if it's there
00030  IF O5 = -1 THEN PRINT"ILLEGAL FILENAME"
      :GOTO 100                              * main menu?
```

```
00040  IF O5 > 0 THEN PRINT"FILE ALREADY EXISTS"
      :INPUT"OK TO WRITE OVER"I$
      :IF I$ <> "Y" THEN 100
00050  OPEN"O", #1, O9$
       .
       .                                  * more instructions
       .
30000  END
```

Subroutine Listing

```
42399  REM filexist
42400  O2$ = O9$ + ":O"
      :O1 = INSTR (O2$, ":")
      :O5 = VAL (MID$ (O2$, O1 + 1))
      :POKE &HEB, - O5 * (O5 < 4 AND O5 > = 0)
      :O2$ = LEFT$ (O2$, O1 - 1)
      :O1 = INSTR (O2$, "/")
      :IF O1 = 0 THEN O1 = INSTR (O2$, ".")
      :IF O1 = 0 THEN O1 = LEN (O2$) + 1
      :O2$ = O2$ + ".DAT"
42410  IF O1 > 9 OR LEN (O2$) - O1 > 3 OR LEN (O2$) > 12
       THEN O5 = - 1
      :RETURN
      :ELSE O3$ = STRING$ (11, 32)
      :MID$ (O3$, 1) = LEFT$ (O2$, O1 - 1)
      :MID$ (O3$, 9) = MID$ (O2$, O1 + 1)
      :FOR T = 1 TO 11
      :POKE T + &H94B, ASC (MID$ (O3$, T, 1))
      :NEXT
      :EXEC &HC65F - 45 * (PEEK (&HC142) < > 48)
      :O5 = PEEK (&H973)
      :RETURN
```

HRINPUT (GOSUB 42500)

Purpose

This subroutine, designed to be used in conjunction with HRCHRSET, allows the input of text from the keyboard on the high resolution graphics screen. The text entered can be edited by positioning a non-destructive cursor (controlled by the left and right arrow keys) over the letter to be changed. The screen is divided into 19 lines and 32 columns for the pupose of start point positioning.

This input routine will only allow text to be entered on the line specified by the calling routine. Line wraparound is not implemented.

The routine uses the powerful graphics commands GET, PUT and DRAW. In normal operation, you need to slow down your typing speed slightly to avoid letters being missed, but, if your computer accepts the speed-up POKE (POKE 65495,0), the routine will even keep up with touch typists.

If you do not need to have a string returned (e.g. if you are using this routine to label a graph), an increase in speed can be achieved by deleting the ":MID$ (O1$,O8,O4) = O2$" in line 42540.

Entry Requirements

1. O1 must specify the screen line (1 to 19) at which the input is to start.

2. O2 must specify the column position (0 to 31) at which the input is to start.

3. O3 must contain the maximum allowable length of the input. O3 can not exceed 32 - O2.

4. The array variables O9() and O8(), used for GET/PUT must have been DIMensioned to 1 and the string array O0$() containing the DRAW information for the graphics character set must have been initialized.

Note: A fractional number may be used for O1 or O2 if you wish to start the input at a "nonstandard point".

Exit Conditions

1. O1$ contains the text entered.

2. O1 and O2 are altered.

Variables Used

O1, O2, O3, O4, O5, O6, O7, O8, O9, O1$, O2$, O3$, O4$, O5$, O6$, O8(), O9(), O0$().

Sample Call

```
00010  PCLEAR 4                          * set up graphics
00020  PMODE 4
00030  SCREEN 1, 0
00040  COLOR 0, 1
00050  DIM O0 (xx)                       * for character set
00060  GOSUB 42700                       * get character set
00070  DIM O9 (1), O8 (1)                * dim for GET/PUT
00080  O1 = 10                           * line 10
00090  O2 = 5                            * column 6
00100  O3 = 12                           * 12 characters
00110  GOSUB 42500                       * get some input
00120  PRINT O1$                         * display results
       .
       .                                 * more instructions
       .
30000 END
```

Subroutine Listing

```
42499  REM hrinput
42500  O3$ = CHR$ (8)
      :O4$ = CHR$ (9)
      :O5$ = " "
      :O6$ = CHR$ (13)
      :O4 = 1
      :O5 = 2
      :O6 = 6
      :O7 = 7
      :O9 = 8
      :O2 = O2 * O9 + O4
      :O1 = O1 * 10 - O5
      :O8 = O4
      :O1$ = STRING$ (O3,32)
```

```
42510  GET (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), O8, G
      :LINE (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), PRESET ,
       BF
      :GET (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), O9, G
      :PUT (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7),O8, PSET
42520  GET (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), O8
      :PUT (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), O8, NOT
42530  O2$ = INKEY$
      :IF O2$ = "" THEN 42530
      :ELSE PUT (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), O8
      :IF O2$ = O6$ THEN RETURN
      :ELSE IF O2$ = O3$ AND O8 > O4 THEN O8 = O8 - O4
      :O2 = O2 - O9
      :POKE &H157, &HFF
      :GOTO 42520
      :ELSE IF O2$ = O4$ AND O8 < O3 THEN O8 = O8 + O4
      :O2 = O2 + O9
      :POKE &H158, &HFF
      :GOTO 42520
      :ELSE IF O2$ < O5$ THEN 42520
42540  PUT (O2 - O4, O1 + O5) - (O2 + O6, O1 - O7), O9, PSET
      :DRAW "BM=O2;,=O1;" + O0$(ASC (O2$) - &H20)
      :MID$ (O1$, O8, O4) = O2$
      :O2 = O2 - ((O8 < O3) * O9)
      :O8 = O8 - (O8 < O3)
      :GOTO 42520
```

HRPRINT (GOSUB 42600)

Purpose

This subroutine, designed to be used in conjunction with HRCHRSET, allows text to be displayed on the high resolution graphics screen. For the purpose of this routine, the screen has been divided into 19 rows with 32 columns each. When the routine is called, the starting row/column must be specified, but, if the line to be printed is longer than the screen width allows, printing will continue at the start of the next screen line.

Entry Requirements

1. O1 contains the line number (1 - 19).

2. O2 contains the column position (0 - 31).

3. O1$ contains the text to be printed.

4. The string array O0$() containing the DRAW information for the graphic character set must have been initialized in advance.

Exit Conditions

O1 and O2 are modified.

Variables Used

O1, O2, O3, O1$, O0$().

Sample Call

```
00010  PCLEAR 4                          * set up graphics
00020  PMODE 4
00030  SCREEN 1, 0
00040  COLOR 0, 1
00050  DIM O0 (xx)                       * for character set
00060  GOSUB 42700                       * get character set
00070  O1$ = "Text on a high resolution screen"
00080  O1 = 8                            * line #
00090  O2 = 0                            * column #
00100  GOSUB 42600                       * go display it
00110  GOTO 110                          * endless loop
       .
       .                                 * more instructions
       .
30000 END
```

Subroutine Listing

```
42599  REM hrprint
42600  O1 = O1 * 10 - 1
      :O2 = O2 * 8 + 1
      :FOR O3 = 1 TO LEN (O1$)
      :DRAW "BM=O2;,=O1;" + OO$(ASC (MID$ (O1$, O3, 1)) - &H20)
      :O2 = O2 + &H8
      :IF O2 > &HFF THEN O2 = &H1
      :O1 = O1 + &HA
      :NEXT
      :RETURN
      :ELSE NEXT
      :RETURN
```

HRCHRSET (GOSUB 42700)

Purpose

Often it is desirable to display text on a high resolution graphics screen. The character set used here is contained in 95 strings which can be displayed with the DRAW statement. The size of the letters can be altered by using the scale argument in DRAW and the direction of printing can be changed with the ANGLE argument (both these options are illustrated in the sample call below). If you use the SCALE option be sure to use only multiples of 4, otherwise uneven lettering will result.

This character set contains all 95 printable ASCII characters. If you do not need all of them for your routine, delete the unneeded lines. The string array numbers correspond to the ASCII representation of the letter minus 32. For example, the data to DRAW the letter "A" (CHR$(65)) is contained in string 65 - 32, or 33. The ASCII characters 123 to 127 are not available from the keyboard. They have been configured as follows:

ASCII VALUE	CHARACTER
123	left brace
124	vertical bar
125	right brace
126	tilde
127	right arrow

The sample program below displays all 95 characters in each of the five graphic modes. You will note that the normal size letters are only readable in PMODE 4. The readability of double-size letters depends on the color set selected in the lower resolution modes. By using a scale factor of 16 an even larger and more readable letter will be displayed. This could be very useful in a program designed for young children just learning to recognize letters and numbers, or for visually handicapped adults.

Entry Requirements

1. The array OO$() must have been DIMensioned to the number of characters in character set.

Exit Conditions

None.

Sample Call

```
00010  REM ************************
00020  REM demonstration program to
00030  REM show various combinations
00040  REM of hrchrset
00050  REM ************************

00060  DIM OO$(95)
00070  GOSUB 42700
00080  DIM GR(11)

00090  REM select graphics mode

00100  FOR PM = 4 TO 0 STEP - 1
00110  RESTORE
00120  PMODE PM
       :SCREEN 1,0
       :COLOR 1,0
       :PCLS 1

00130  REM black border

00140  LINE (10,10) - (245,10), PRESET
00150  LINE - (245,181), PRESET
00160  LINE - (10,181), PRESET
00170  LINE - (10,10), PRESET
00180  PAINT (0,0),0

00190  REM print top line

00200  DRAW "S4"
       :X = 20
       :Y = 8
       :BL$ = "BR6"
       :READ TX$
       :GOSUB 340

00210  REM shift top of line right and bottom
00220  REM left to achieve italics effect

00230  GET (20,2) - (245,3),GR,G
       :PUT (21,2) - (246,3),GR, PSET
00240  GET (20,7) - (245,8),GR,G
       :PUT (19,7) - (244,8),GR, PSET
```

```
00250  REM right side message going down
00260  DRAW "A1"
      :X = 247
      :Y = 4
      :BL$ = "BR3"
      :READ TX$
      :GOSUB 340

00270  REM bottom message upside-down

00280  DRAW "A2"
      :X = 230
      :Y = 183
      :BL$ = "BR5"
      :READ TX$
      :GOSUB 340

00290  REM left side message going up

00300  DRAW "A3"
      :X = 9
      :Y = 186
      :BL$ = "BR3"
      :READ TX$
      :GOSUB 340
00310  DRAW "A0"
      :COLOR 0,1
00320  GOTO  360

00330  REM subroutine to display TX$ one letter at a time

00340  DRAW "BM=X;,=Y;"
      :FOR T = 1 TO LEN (TX$)
      :DRAW  OO$(ASC (MID$ (TX$,T,1)) - 32) + BL$
      :NEXT T
      :RETURN

00350  REM display entire character set, normal size

00360  Y = 24
      :X = 20
      :FOR T = 0 TO 95
      :DRAW "BM=X;,=Y;XOO$(T);"
      :X = X + 10
      :IF X > 232 THEN X = 20
      :Y = Y + 9
00370  NEXT  T
```

```
00380  REM display entire character set in double size

00390  DRAW "S8"
      :X = 19
      :Y = Y + 18
      :FOR T = 0 TO 95
      :DRAW "BM=X;,=Y;XOO$(T);"
      :X = X + 16
      :IF X > 232 THEN X = 19
      :Y = Y + 16
00400  NEXT T

00410  REM do next color set and pmode

00420  FOR T = 1 TO 1000
      :NEXT T
00430  SCREEN 1,1
      :FOR T = 1 TO 2000
      :NEXT T
00440  NEXT PM
00450  RUN

00460  REM data for messages

00470  DATA  GRAPHICS CHARACTER SET,TEXT CAN RUN DOWN THE
       SIDE,* THIS IS UPSIDE DOWN *,TEXT CAN RUN UP THE
       SCREEN

30000  END
```

Subroutine Listing

```
42698  REM hrchrset

42699  REM SPACE - "/"
42700  OO$(0) = "BR4"
      :OO$(1) = "BR2UBU2U3BR2BD6"
      :OO$(2) = "BRBU4U2BR2D2BRBD4"
      :OO$(3) = "BRBUUNLNR3U2NLNUR2NUNRD3BDBR"
      :OO$(4) = "BUR2DUREHL2HERUDR2BD5"
      :OO$(5) = "BUE4BL3LURDBR3BD4NDLDR"
      :OO$(6) = "BR4BU2G2LHE3UHLGDF4"
      :OO$(7) = "BR2BU4U2BR2BD6"
```

```
42710  OO$(8) = "BU6BR2NRGD4FRBR"
      :OO$(9) = "REU4HNLBR3BD6"
      :OO$(10) = "BUE2NL2NH2NU2NE2ND2NF2R2BD3"
      :OO$(11) = "BU3R2NU2ND2R2BD3"
      :OO$(12) = "BR3BULURD2GBR2BU"
      :OO$(13) = "BRBU3R2BRBD3"
      :OO$(14) = "BR2LURDBR2"
      :OO$(15) = "UE4UBD6"

42719  REM "0" - "9"
42720  OO$(16) = "BUNE4U4ER2FD4GL2HBR4BD"
      :OO$(17) = "BRBU5ED6LR2BR"
      :OO$(18) = "BU5ER2FDG4R4"
      :OO$(19) = "BU5ER2FDGLRFDGL2HBR4BD"
      :OO$(20) = "BR3U6G3R4BD3"
42730  OO$(21) = "BUFR2EUHL3U3R4BD6"
      :OO$(22) = "BUNFU4ER2FBD2BLNL3FDGNLBR"
      :OO$(23) = "BU6R4G3D3BR3"
      :OO$(24) = "BR4BU2DGL2HUER2L2HUER2FDGFBD2"
      :OO$(25) = "BUFR2EU4HL2GDFR2BRBD3"

42739  REM ":" - "?"
42740  OO$(26) = "BRBURULDBU3RULDBR3BD4"
      :OO$(27) = "BR2BULURD2GBRBU5LURDBR2BD4"
      :OO$(28) = "BR4BU6G3F3"
      :OO$(29) = "BU4BRR2BD2NL2BRBD2"
      :OO$(30) = "E3H3BR4BD6"
      :OO$(31) = "BU5ER2FDGLDBDDBR2"
      :OO$(32) = "BUNFU4ER2FDL2GFR2NU2BD2NL2"

42749  REM "A" - "Z"
42750  OO$(33) = "U4E2F2D2NL4D2"
      :OO$(34) = "RU6LR3FDGNL2FDGL3BR4"
      :OO$(35) = "BUNFU4ER2FDBD2DGNL2BR"
      :OO$(36) = "RU6LR3FD4GL2BR3"
      :OO$(37) = "U3NR3U3R4BD6NL4"
      :OO$(38) = "U3NR3U3R4BD6"
      :OO$(39) = "BR2BU3R2D2GL2HU4ER2FBD5"
      :OO$(40) = "U6D3R4U3D6"
42760  OO$(41) = "BRR2LU6LR2BRBD6"
      :OO$(42) = "BU2DFR2EU5BD6"
      :OO$(43) = "U6BR4G3F3"
      :OO$(44) = "NR4U6BR4BD6"
      :OO$(45) = "U6F2E2D6"
      :OO$(46) = "U6DF4U5D6"
      :OO$(47) = "R4L4U6R4D6"
      :OO$(48) = "U6R3FDGL3BR4BD3"
      :OO$(49) = "BUU4ER2FD4GL2HBR2BU1F2"
```

```
42770  OO$(50) = "U6R3FDGL3RF3"
      :OO$(51) = "BUFR2EUHL2HUER2FBD5"
      :OO$(52) = "BU6R4L2D6BR2"
      :OO$(53) = "U6D6R4U6D6"
      :OO$(54) = "BU6D4F2E2U4BD6"
      :OO$(55) = "U6D6E2F2U6D6"
      :OO$(56) = "UE2H2UDF2E2UDG2F2D"
      :OO$(57) = "BU6DF2E2UDG2D3BR2"
      :OO$(58) = "NR4UE4UNL4BD6"

42779  REM "[" - "@"
42780  OO$(59) = "BRU6NR2D6R2BR"
      :OO$(60) = "BU6DF4D"
      :OO$(61) = "BRR2U6NL2D6BR"
      :OO$(62) = "BU4E2ND6F2BD4"
      :OO$(63) = "BU3NF2NE2R4BD3"
      :OO$(64) = "BU2F2NU6E2BD2"

42789  REM "a" - "z"
42790  OO$(65) = "BUNFNEBU3R3FDNL3D2NL3"
      :OO$(66) = "U6D2R3FD2GNL2BR"
      :OO$(67) = "BRNR2HU2ER2FBD2GBR"
      :OO$(68) = "BRNR3HU2ER3U2D6"
      :OO$(69) = "BRNR3HU2ER2FDNL3BD2"
      :OO$(70) = "BR2U3NRNLU2EFBD5"
      :OO$(71) = "BRNR2HU2ER3D5GNL3EU"
      :OO$(72) = "U4NU2R3FD3"
42800  OO$(73) = "BRRNRU3NLBU2UBR2BD6"
      :OO$(74) = "BDFR2EU4BU3DBD2D3"
      :OO$(75) = "BRU2NU4E3BD5NH3"
      :OO$(76) = "BRRNRU6NLBR2BD6"
      :OO$(77) = "U4F2NDE2D4"
      :OO$(78) = "U4R3FD3"
      :OO$(79) = "BRNR2HU2ER2FD2GBR"
      :OO$(80) = "D2U6R3FD2GNL3BR"
      :OO$(81) = "BRNR3HU2ER3D6U2"
42810  OO$(82) = "U4DER2FBD3"
      :OO$(83) = "BU4BR4L3GFR2FGNL3BR"
      :OO$(84) = "BRNR2U4NRNLU2BR3BD6"
      :OO$(85) = "BRHU3BR3D4NL2R"
      :OO$(86) = "BU4DF2NDE2NUBD3"
      :OO$(87) = "NU4E2F2NU4"
      :OO$(88) = "E2NH2NE2F2"
      :OO$(89) = "BU3NUF2DGNDE3NU2BD2"
      :OO$(90) = "BU4R4G4R4"
```

```
42819  REM CHR$(123) - CHR$(127)
42820  OO$(91) = "BR2NRHU2NLU2ERBRBD6"
      :OO$(92) = "BR2U2BU2U2BR2BD6"
      :OO$(93) = "BRREU2NRU2HNLBR2BD6"
      :OO$(94) = "BU4EUF2ENUBU5"
      :OO$(95) = "BU3R4NH2NG2BD3"
42830  RETURN
```

PCLEAR0 (GOSUB 63900)

Purpose

This subroutine compensates for the lack of a PLCEAR 0 command in Extended COLOR BASIC. You can now free up that last 1536 bytes of graphics memory for your BASIC program. If you have a disk system and need to use the FILES command, do so before using this subroutine.

When this routine is called it first checks to see if the beginning of the BASIC program area is the same as the beginning of the graphics area. If it is, a RETURN is done; if not, the following machine language routine is poked into memory at &H01DD.

```
CLRB            clear carry and
INCB            set lsb of new start
LDA     $BC     D = start of graphics + 1
TFR     D,Y     copy D into Y
JMP     $96A3   jump to PCLEAR ROM routine
```

Try to keep this routine near the end of your program; otherwise ?SN ERRORS may occur in the same manner they occur with other PCLEAR statements. To assist in placement we have departed from our normal numbering system and used line 63900 for PCLEAR0.

Entry Requirements

1. The call to PCLEAR0 should be in the first line of the BASIC program.

Exit Conditions

1. No graphics memory is reserved for high resolution graphics.

CAUTIONARY NOTE: DO NOT USE ANY EXTENDED COLOR BASIC GRAPHICS COMMANDS AFTER THIS ROUTINE HAS BEEN EXECUTED. If you do, a part, or all, of the BASIC program in memory will be destroyed. Extended Color BASIC still presumes that the first grahics page is available for grahics use. Since part of your program is now residing in this area, graphics commands will affect the program code.

Variables Used

O1, O2, O1$.

Sample Call

```
00010  GOSUB 63900                          * call PCLEAR0
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listing

```
63899  REM pclear0
63900  IF PEEK (25) * 256 + PEEK (26) = PEEK (188) * 256 +
       PEEK (189) + 1 THEN RETURN
      :ELSE O1 = &HO1DD
      :O1$ = "5F5C96BC1F027E96A3"
      :FOR O2 = 1 TO 18 STEP 2
      :POKE O1, VAL ("&H" + MID$ (O1$, O2, 2))
      :O1 = O1 + 1
      :NEXT
      :EXEC &HO1DD
      :RUN
```

Section Four

MACHINE LANGUAGE SUBROUTINES

MACHINE LANGUAGE SUBROUTINES

This section of the text contains several machine language subroutines which can be used by a BASIC program for special screen effects, for speeding up keyboard input, or for a variety of other purposes.

Each of the routines has been written in position independent code, which means that your BASIC program can place the required routines anywhere in RAM memory and they will still operate correctly. As you will see when you browse through this section, each machine language routine has been converted into at least two BASIC routines, a primary initialization routine and one or more secondary routines which call the different entry points of the machine language program to invoke its different functions.

The primary BASIC routine (e.g. INPUT INIT) contains a short segment which determines whether or not the machine language data has already been POKEd into the desired memory locations. If not, then one or more data strings of the following type,

```
O1$ = "8634B6FF23....",
```

where each pair of characters represents a hexadecimal number between 0 and 255, is parsed by another short segment in the initializing routine, and the data is stored in memory. The final segment then sets up a definition for the required USR call.

The secondary BASIC routine(s) (e.g. INPUT MAIN) make the actual call to the proper machine language entry point, ensuring that any necessary parameters are passed in the correct format. In most cases, the call is made via BASIC's USR statement, but occasionally (as in the CLOCK and TIMER routines), a simple EXEC is enough to obtain the desired results.

This approach to incorporating machine language subroutines in a BASIC program simplifies the requirements for you if you wish to include them in your program. In all cases, all you have to do is make a GOSUB call to the correct program line, and our BASIC subroutine will take care of everything. In some cases, of course, you will have to ensure that a variable has been properly defined before you actually make the call.

For the sake of uniformity, we have set up all of the examples in this section so that the machine language code is POKEd into memory starting at address &HO1DD (the cassette I/O buffer). When you decide to use these routines, please realize

that you are NOT required to store the routines in the same area. In fact, if you decide to use more than one routine in your program (which could cause the total size of all the routines to exceed the size of the buffer), you may have to find a different unused block of RAM to store the code. In addition, because cassette input/output causes data to be written into the I/O buffer, you cannot use this buffer for both purposes simultaneously.

You should also bear in mind that most of the routines make use of the USR function in order to pass parameters back and forth; all of the examples (that uniformity again) define USR 0 through USR 2. Obviously, if you decide to include more than one routine in your program, you may have to change the USR definitions (i.e. USR 3 through USR 9) for one or more of them so that independent USR calls will still be possible.

A special subroutine, MLSET, allows you to quickly POKE several other routines into a contiguous block of memory. Once the routines have been initialized, the USR function called by MLSET can be re-defined for use by any other routine.

As in the BASIC subroutines in the preceding section, we have restricted ourselves here to variable names which begin with the letter "O", simply because such variables don't seem to find their way into BASIC programs too often. This will not pose any problems to existing programs unless you are using such variable names to represent constant values. If this is the case, we suggest that such variable names be changed to avoid any conflict. In any event, the variables that we use need only be defined while our subroutines are actually active. Following the RETURN, you are free to use these variables for any other purpose, if you wish.

INPUT INIT (GOSUB 50000)
INPUT MAIN (GOSUB 50100)

Purpose

This subroutine allows you to obtain data from the keyboard. It works in exactly the same way as BASIC's INPUT statement, except that the BREAK key does not have any effect on program execution. Instead, the BREAK key is treated exactly like the ENTER key and causes keyboard input to be terminated. All of BASIC's standard editing keys are still active (left arrow, shift left arrow, and CLEAR) and will produce exactly the same response that you are used to.

The routine uses a very neat trick for passing up to 255 characters of string data to the BASIC program: it creates a temporary string descriptor which points to BASIC's normal input buffer at memory location &H02DD. On return from the subroutine, the string is forced into the string pool in upper memory by the simple concatenation of a null string.

Assembly Language Listing

```
                        ********************************
                        *     SIMPLE INPUT ROUTINE     *
                        ********************************

0E00                            ORG $01DD       in the cassette buffer
01DD                    INPUT   EQU *
01DD BDB3ED                     JSR $B3ED       get descriptor addr
01E0 1F02                       TFR D,Y         need A & B, so use Y
01E2 DE88                       LDU $88         get cursor address
01E4 BDA390                     JSR $A390       go get some input
01E7 5A                         DECB            adjust to true length
01E8 E7A4                       STB ,Y          save len in descriptor
01EA 2602                       BNE IN02        not zero length, skip
01EC DF88                       STU $88         reset cursor address
01EE                    IN02    EQU *
01EE CC02DD                     LDD #$02DD      get buffer address
01F1 ED22                       STD 2,Y         save in strg descriptor
01F3 39                         RTS             all done; return

01F4                            END INPUT

NO ERRORS FOUND
```

Entry Requirements--INPUT INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--INPUT INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--INPUT MAIN

None.

Exit Conditions--INPUT MAIN

1. The resulting input is returned in variable 01$. Your main program must transfer the data to another string of your own choice in order to protect the original data from being over-written during subsequent calls. This can be accomplished by a simple statement like

A$ = 01$.

Variables Used

00, 01, 01$, 02$.

Sample Call

```
00010. CLEAR 1000                          * lots of string space
      :00 = &H01DD                         * set start address
      :GOSUB 50000                         * go init subroutine
      :CLS
      :PRINT "ENTER SOME DATA? ";
00020  GOSUB 50100                         * go get some input
      :A$ = 01$                            * transfer the data
00030  CLS
      :PRINT "ENTER SOME MORE DATA? ";
00040  GOSUB 50100
      :B$ = 01$
```

```
00050  PRINT A$
      :PRINT B$
      :PRINT O1$
       .
       .                                  * more instructions
       .
30000  END
```

Subroutine Listings

```
49999  REM input init
50000  IF PEEK (OO) = &HBD AND PEEK (OO + 22) = &H39 THEN
       RETURN
      :ELSE O1$ = "BDB3ED1F02DE88BDA3905AE7A42602DF88CC02DD
       ED2239"
      :FOR O1 = 0 TO 22
      :O2$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :POKE OO + O1, VAL (O2$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN

50099  REM input main
50100  O1$ = ""
      :O1 = USR 0 (VARPTR (O1$))
      :O1$ = O1$ + ""
      :RETURN
```

FLASH INIT (GOSUB 50200)
FLASH MAIN (GOSUB 50300)

Purpose

This subroutine performs two functions simultaneously: it causes all graphic characters on the screen to change color, and it provides a flashing cursor while it waits for a single keystroke. It is ideally suited for use as a menu selection input routine, since exit does not occur until the key pressed matches one of a predefined set of allowed characters.

Assembly Language Listing

```
                    ************************************
                    * FLASHING SCREEN/SINGLE KEY INPUT *
                    ************************************

0E00                        ORG $01DD       in cassette buffer

01DD                FLASH   EQU *
01DD BDB3ED                 JSR $B3ED       get descriptor addr
01E0 1F02                   TFR D,Y         need A & B, so use Y
01E2 E6A4                   LDB ,Y          get string length
01E4 2740                   BEQ FL08        invalid string, leave
01E6 10AE22                 LDY 2,Y         get actual string addr
01E9 3424                   PSHS B,Y        save both
01EB DE88                   LDU $88         get cursor address

01ED                FL01    EQU *
01ED 0A94                   DEC $94         time to flash colors?
01EF 2612                   BNE FL04        no, skip
01F1 8E0400                 LDX #$400       get screen start addr

01F4                FL02    EQU *
01F4 A684                   LDA ,X          get screen byte
01F6 2A04                   BPL FL03        not graphic, skip
01F8 8B10                   ADDA #$10       next color
01FA 8A80                   ORA #$80        force graphic

01FC                FL03    EQU *
01FC A780                   STA ,X+         byte to screen & bump
01FE 8C0600                 CMPX #$600      end of screen yet?
0201 25F1                   BLO FL02        loop til done
0203                FL04    EQU *
0203 9694                   LDA $94         get cursor delay count
0205 857F                   BITA #$7F       time to flash cursor?
0207 2606                   BNE FL05        no, skip
0209 A6C4                   LDA ,U          get cursor byte
```

```
020B 8840              EORA #$40       invert
020D A7C4              STA ,U          and put it back

020F           FL05    EQU *
020F BDA1C1            JSR $A1C1       go get a keypress
0212 27D9              BEQ FL01        no key, loop back
0214 E6E4              LDB ,S          get string length
0216 10AE61            LDY 1,S         get address

0219           FL06    EQU *
0219 A1A0              CMPA ,Y+        acceptable key?
021B 2705              BEQ FL07        yes, leave
021D 5A                DECB            tested entire string?
021E 26F9              BNE FL06        loop til done
0220 20CB              BRA FL01        invalid key, restart

0222           FL07    EQU *
0222 3524              PULS B,Y        cleanup stack
0224 1F89              TFR A,B         get LSB of key value

0226           FL08    EQU *
0226 4F                CLRA            set MSB to 0
0227 7EB4F4            JMP $B4F4       to user and return

022A                   END FLASH

NO ERRORS FOUND
```

Entry Requirements--FLASH INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--FLASH INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--FLASH MAIN

1. Variable 01$ must be pre-defined to contain the list of allowable characters. The routine uses this list as a comparison base when a key is pressed. Exit does not occur until a match is found.

Exit Conditions--FLASH MAIN

1. The ASCII value of the key that was actually pressed is returned in variable O1.

2. The list that was passed in variable O1$ is retained in its original form.

Variables Used

O0, O1, O1$, O2$.

Sample Call

```
00010  CLS 0                                  * fancy screen
      :V = &H85
      :FOR I = 0 TO 31 STEP 2
      :POKE 1024 + I, V
      :POKE 1025 + I, V
      :POKE 1534 - I, V
      :POKE 1535 - I, V
      :V = V + &H10
      :IF V > &HF5 THEN V = &H85
00020  NEXT
      :V = &H86
      :FOR I = 1 TO 14
      :POKE 1024 + 32 * (15 - I), V + 3
      :POKE 1025 + 32 * (15 - I), V
      :POKE 1054 + 32 * I, V
      :POKE 1055 + 32 * I, V + 3
      :V = V + &H10
      :IF V > &HE6 THEN V = &H86
00030  NEXT
00040  PRINT @232, "THIS IS A TEST  " + CHR$ (8);
      :SCREEN 0,1
      :O0 = &H01DD                           * set start address
      :GOSUB 50200                           * go init subroutine
      :O1$ = CHR$ (13)                       * wait for <ENTER>
      :GOSUB 50300                           * go do it
00050  CLS
       .                                     * more instructions
       .
       .
30000  END
```

Subroutine Listings

```
50199  REM flash init
50200  IF PEEK (00) = &HBD AND PEEK (00 + 76) = &HF4 THEN
       RETURN
      :ELSE 01$ = "BDB3ED1F02E6A4274010AE223424DE880A942612
       8E0400A6842A048B108A80A7808C060025F19694857F2606A6C4
       8840A7C4BDA1C127D9E6E410AE61A1A027055A26F920CB3524
       1F894F7EB4F4"
50210  FOR 01 = 0 TO 76
      :02$ = "&H" + MID$ (01$, 2 * 01 + 1, 2)
      :POKE 00 + 01, VAL (02$)
      :NEXT
      :DEF USR 0 = 00
      :RETURN

50299  REM flash main
50300  01$ = 01$ + ""
      :01 = USR 0 (VARPTR (01$))
      :POKE &H94, 1
      :RETURN
```

SCROLL INIT (GOSUB 50400)
SCROLL MAIN (GOSUB 50500)

Purpose

This subroutine allows you to cause scrolling of the text screen in one of four possible directions--up, down, left, or right. When upward scrolling is selected, the data on the screen moves upward one line and the bottom line is cleared in preparation for new data to be printed. Similarly, downward scrolling causes the topmost line to be cleared to allow for immediate printing. Horizontal scrolling is a little different, however, since no data is actually removed from the screen. Instead the leftmost or rightmost column of data is moved to the opposite end of the screen, depending on which direction is selected.

Assembly Language Listing

```
                    **********************************
                    *       SCREEN SCROLLING         *
                    **********************************

0E00                        ORG $01DD          in cassette buffer

01DD                SCROLL  EQU *
01DD BDB3ED                 JSR $B3ED          get descriptor addr
01E0 1F02                   TFR D,Y            need A & B, use Y
01E2 E6A4                   LDB ,Y             get length
01E4 5A                     DECB               length = 1?
01E5 267B                   BNE SCRDUN         no, leave
01E7 A6B802                 LDA [2,Y]          get the character
01EA 8144                   CMPA #'D           down scroll?
01EC 2719                   BEQ DOWN           yes, skip
01EE 814C                   CMPA #'L           left scroll?
01F0 2731                   BEQ LEFT
01F2 8152                   CMPA #'R           right scroll?
01F4 274D                   BEQ RIGHT
01F6 CE0400                 LDU #$0400         dest addr = top line
01F9 30C820                 LEAX $20,U         source = 1 line below

01FC                UP      EQU *
01FC EC81                   LDD ,X++           2 bytes at a time
01FE EDC1                   STD ,U++           move up a line
0200 8C0600                 CMPX #$0600        end of screen?
0203 25F7                   BLO UP             loop til done
0205 2011                   BRA LASTLN         clear top or bottom
0207                DOWN    EQU *
0207 CE0600                 LDU #$0600         dest address
```

```
020A 30C8E0            LEAX -$20,U     source address

020D           DN01    EQU *
020D EC83              LDD ,--X        2 bytes at a time
020F EDC3              STD ,--U        move down a line
0211 8C0400            CMPX #$0400     top of screen yet?
0214 22F7              BHI DN01        loop til done
0216 1F13              TFR X,U         addr for last line

0218           LASTLN  EQU *
0218 DF88              STU $88         new cursor address
021A CC6020            LDD #$6020      A = blank; B = count

021D           LST01   EQU *
021D A7C0              STA ,U+         clear last line
021F 5A                DECB
0220 26FB              BNE LST01       loop til done
0222 39                RTS             all done; leave

0223           LEFT    EQU *
0223 CE0400            LDU #$400       start of screen
0226 3041              LEAX 1,U        one position to right

0228           LF01    EQU *
0228 E6C4              LDB ,U          get 1st char on line
022A 3404              PSHS B          save til end of line
022C C61F              LDB #$1F        31 bytes to move

022E A680      LF02    LDA ,X+         get a source byte
0230 A7C0              STA ,U+         move left
0232 5A                DECB            done all bytes?
0233 26F9              BNE LF02        loop for whole line
0235 3001              LEAX 1,X        adjust addresses
0237 3341              LEAU 1,U
0239 3504              PULS B          get 1st byte
023B E75F              STB -1,U        put at end of line
023D 8C0600            CMPX #$600      end of screen yet?
0240 25E6              BLO LF01        loop for whole screen
0242 39                RTS             done, leave

0243           RIGHT   EQU *
0243 CE0600            LDU #$600       end of screen
0246 305F              LEAX -1,U       previous position
0248           RT01    EQU *
0248 E65F              LDB -1,U        last byte on line
024A 3404              PSHS B          save til beginning
024C C61F              LDB #$1F        31 chars to move
```

```
024E               RTO2    EQU *
024E A682                  LDA ,-X          get a byte
0250 A7C2                  STA ,-U          move it right
0252 5A                    DECB             done all bytes?
0253 26F9                  BNE RTO2         loop for whole line
0255 301F                  LEAX -1,X        adjust addresses
0257 335F                  LEAU -1,U
0259 3504                  PULS B           get last char
025B E7C4                  STB ,U           put it at beginning
025D 8C0400                CMPX #$400       top of screen yet?
0260 22E6                  BHI RTO1         loop for whole screen

0262               SCRDUN  EQU *
0262 39                    RTS

0263                       END SCROLL
```

NO ERRORS FOUND

Entry Requirements--SCROLL INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable OO.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--SCROLL INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--SCROLL MAIN

1. Variable O1$ must be defined as a single character string which tells the routine which direction to scroll. The string must contain one of the following upper case letters:

   a. U -- up
   b. D -- down
   c. L -- left
   d. R -- right

Exit Conditions--SCROLL MAIN

1. The data on the screen will be moved once in the direction specified by O1$.

2. The contents of O1$ are preserved; this allows for repeated scrolling in one direction without re-defining the string.

Variables Used

OO, O1, O1$, O2$, O3$, O4$.

Sample Call

```
00010  CLS 0                                  * fancy screen
      :V = &H85
      :FOR I = 0 TO 31 STEP 2
      :POKE 1024 + I, V
      :POKE 1025 + I, V
      :POKE 1534 - I, V
      :POKE 1535 - I, V
      :V = V + &H10
      :IF V > &HF5 THEN V = &H85
00020  NEXT
      :V = &H86
      :FOR I = 1 TO 14
      :POKE 1024 + 32 * (15 - I), V + 3
      :POKE 1025 + 32 * (15 - I), V
      :POKE 1054 + 32 * I, V
      :POKE 1055 + 32 * I, V + 3
      :V = V + &H10
      :IF V > &HE6 THEN V = &H86
00030  NEXT
00040  PRINT @233, "THIS IS A TEST";
      :OO = &H01DD                           * set start address
      :GOSUB 50400                           * go init subroutine
```

```
00050  FOR J = 1 TO 16                    * multiple loops
      :O1$ = "L"                          * for special
      :FOR I = 1 TO 32                    * scrolling effects
      :GOSUB 50500
      :NEXT I
      :O1$ = "D"
      :GOSUB 50500
      :NEXT J
      :CLS
      .                                   * more instructions
      .
      .
30000  END
```

Subroutine Listings

```
50399  REM scroll init
50400  IF PEEK (OO) = &HBD AND PEEK (OO + 133) = &H39 THEN
       RETURN
      :ELSE O1$ = "BDB3ED1F02E6A45A267BA6B80281442719814C
       27318152274DCE040030C820EC81EDC18C060025F72011CE0600
       30C8E0EC83EDC38C040022F71F13DF88CC6020A7C05A"
50410  O2$ = "26FB39CE04003041E6C43404C61FA680A7C05A26F9
       300133413504E75F8C060025E639CE0600305FE65F3404C61F
       A682A7C25A26F9301F335F3504E7C48C040022E639"
50420  FOR O1 = 0 TO 66
      :O3$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :O4$ = "&H" + MID$ (O2$, 2 * O1 + 1, 2)
      :POKE OO + O1, VAL (O3$)
      :POKE OO + O1 + 67, VAL (O4$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN

50499  REM scroll main
50500  O1$ = O1$ + ""
      :O1 = USR 0 (VARPTR (O1$))
      :RETURN
```

TIMER INIT	(GOSUB 50600)
ENABLE TIMER	(GOSUB 50700)
SET START TIME	(GOSUB 50800)
GET TIME REMAINING	(GOSUB 50900)
DISABLE TIMER	(GOSUB 51000)

Purpose

This is actually one routine with four separate entry points. The routine sets up a timer which can be used within your program to fix a time-limit on a specific operation, which could be useful in an application such as a game or an educational program. It makes use of the 6809's ability to generate a regular interrupt 60 times per second.

Two of the entry points allow you to turn the timer on or off. When the timer is not in use, it should be turned off because, due to the increased length of the interrupt service routine, the processor will be spending quite a bit more time away from your BASIC program; this will result in a marginally slower program.

The SET START TIME entry point allows you to set the clock to any value from 0 to 3600 seconds, which means that your time limit can be any value from one second to one hour. The GET TIME REMAINING entry point does exactly that--returns a value which represents the number of seconds remaining. When the clock reaches zero, the count-down stops; that is, the time remaining will never drop below zero.

Although you may have both the TIMER and the CLOCK (see next chapter) in memory simultaneously, do NOT enable them both simultaneously. This could have disastrous results...

Assembly Language Listing

```
                        ********************************
                        *          TIMING CLOCK        *
                        ********************************

0E00                            ORG $01DD       in cassette buffer

                        * Entry point #1--routine setup

01DD                    SETUP   EQU *
01DD 3401                       PSHS CC         save interrupt status
01DF 1A50                       ORCC #$50       disable IRQ & FIRQ
01E1 308C28                     LEAX <TIMER,PCR get our interrupt addr
01E4 BC010D                     CMPX $10D       already setup?
```

```
01E7 2709              BEQ SU01        yes, leave
01E9 FE010D            LDU $10D        get BASIC's vector
01EC EF8C34            STU <PATCH,PCR  save it for jump
01EF BF010D            STX $10D        save our vector

01F2             SU01  EQU *
01F2 3581              PULS CC,PC      restore & return

                 * Entry point #2--set initial count value

01F4             SETTIM EQU *
01F4 BDB3ED            JSR $B3ED       get user's start count
01F7 10830E10          CMPD #3600      allowable?
01FB 2302              BLS ST01        ok, skip
01FD 4F                CLRA            too big, use zero
01FE 5F                CLRB

01FF             ST01  EQU *
01FF ED8C07            STD <SECS,PCR   save count
0202 39                RTS             and return

                 * Entry point #3--get time remaining for user

0203             GETTIM EQU *
0203 EC8C03            LDD <SECS,PCR   get # of seconds
0206 7EB4F4            JMP $B4F4       give to user & return

                 * Interrupt Servicing Routine

0209 0000        SECS  FDB $0000
020B 00          CYCLES FCB $00

020C             TIMER EQU *
020C 308CFA            LEAX SECS,PCR   point to parameter list
020F 6C02              INC 2,X         bump times through loop
0211 A602              LDA 2,X
0213 813C              CMPA #60        1 second yet?
0215 250B              BLO TM01        no, skip
0217 6F02              CLR 2,X         reset cycle count
0219 EC84              LDD ,X          get seconds
021B 2705              BEQ TM01        already zero, skip
021D 830001            SUBD #1         countdown
0220 ED84              STD ,X          replace value

0222             TM01  EQU *
0222 7E                FCB $7E         "JMP" opcode

0223 0000        PATCH FDB $0000       where to go when done
```

```
                    * Entry point #4--restore BASIC's vector

0225                RSTBAS EQU *
0225 3401                  PSHS CC          save interrupt status
0227 1A50                  ORCC #$50        disable IRQ & FIRQ
0229 EC8CF7                LDD PATCH,PCR    get BASIC's vector
022C 2703                  BEQ RB01         undefined, skip
022E FD010D                STD $10D         restore for BASIC

0231                RB01   EQU *
0231 3581                  PULS CC,PC       restore & return

0233                       END SETUP

NO ERRORS FOUND
```

Entry Requirements--TIMER INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable OT--this variable MUST remain properly defined as long as the timer routine is in memory.

2. The INIT subroutine must be called at least once before any of the other subroutines can be called.

3. Note that the INIT routine automatically falls through to the ENABLE TIMER routine, so you don't have to explicitly call the ENABLE routine following the INIT call.

Exit Conditions--TIMER INIT

1. INIT defines USR 0 and USR 1 for use by the other subroutines.

2. All variables except OT are released for re-use.

Entry Requirements--ENABLE TIMER

None.

Exit Conditions--ENABLE TIMER

None.

Entry Requirements--SET START TIME

1. Variable O1 must contain a numeric value between 0 and 3600.

Exit Conditions--SET START TIME

1. The value in O1 is retained by the routine as the new count-down starting value.

Entry Requirements--GET TIME REMAINING

None.

Exit Conditions--GET TIME REMAINING

1. The time remaining (in seconds) is returned in variable O1.

Entry Requirements--DISABLE TIMER

None.

Exit Conditions--DISABLE TIMER

1. The timer is physically removed from the interrupt service routine. If, later on in your program, you wish to re-use the timer, you can do so by making a call to the ENABLE TIMER routine, which will put the routine back into the interrupt handler.

Variables Used

OT, O1, O1$, O2$.

Sample Call

```
00010  OT = &H01DD                          * set start address
      :GOSUB 50600                          * go INIT & ENABLE
00020  CLS
      :A$ = "####"
      :B$ = "V31T255L255O1CO3CO5C"
      :O1 = 60
      :GOSUB 50800                          * set start time to 60
00030  GOSUB 50900                          * get time remaining
      :IF O1 = 0 THEN 40
      :ELSE PRINT @224, USING A$;O1;
      :IF A = O1 THEN 30
      :ELSE A = O1
      :PLAY B$
      :GOTO 30
```

```
00040  GOSUB 51000                          * disable timer
                                            * more instructions
       .
       .
       .
30000  END
```

Subroutine Listings

```
50599  REM timer init
50600  IF PEEK (OT) = &H34 AND PEEK (OT + 69) = &H7E THEN
       RETURN
      :ELSE O1$ = "34011A50308C28BC010D2709FE010DEF8C34
       BF010D3581BDB3ED10830E1023024F5FED8C0739EC8C037EB4F4
       000000308CFA6C02A602813C250B6F02EC842705830001ED84
       7E00003401 1A50EC8CF72703FD010D3581"
50610  FOR O1 = 0 TO 85
      :O2$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :POKE OT + O1, VAL (O2$)
      :NEXT
      :DEF USR 0 = OT + 23
      :DEF USR 1 = OT + 38

50699  REM enable timer
50700  EXEC OT
      :RETURN

50799  REM set start time
50800  O1 = USR 0 (O1)
      :RETURN

50899  REM get time remaining
50900  O1 = USR 1 (0)
      :RETURN

50999  REM disable timer
51000  EXEC OT + 72
      :RETURN
```

CLOCK INIT	(GOSUB 51100)
ENABLE CLOCK	(GOSUB 51200)
DISABLE CLOCK	(GOSUB 51300)
DISPLAY ON/OFF	(GOSUB 51400)
SET TIME	(GOSUB 51500)
SET DISPLAY POSITION	(GOSUB 51600)

Purpose

This is actually one routine with five separate entry points. The routine sets up a 24-hour clock which can be used in any BASIC program to display the time. The various entry points allow you to enable or disable the clock, turn the display on or off (while leaving the clock running), determine where the time will appear on the screen, and to set the time. Although it is quite safe to have both CLOCK and TIMER in memory at the same time, it is definitely a no-no to have both routines enabled simultaneously.

Assembly Language Listing

```
                    **********************************
                    *        STANDARD CLOCK          *
                    **********************************

0E00                        ORG $01DD       in cassette buffer

                    * Entry point #1--turn clock display on/off

01DD                ONOFF   EQU *
01DD BDB3ED                 JSR $B3ED       get descriptor addr
01E0 1F01                   TFR D,X         need A & B, so use X
01E2 A684                   LDA ,X          get string length
01E4 4A                     DECA            length = 1?
01E5 260B                   BNE 0002        no, leave
01E7 A69802                 LDA [2,X]       get the character
01EA 814F                   CMPA #'O        <O>n flag?
01EC 2701                   BEQ 0001        yes, skip
01EE 4F                     CLRA            no, reset flag

01EF                0001    EQU *
01EF A78C01                 STA <FLAG,PCR   save for intrpt routine

01F2                0002    EQU *
01F2 39                     RTS             return

01F3 00             FLAG    FCB $00
```

```
                    * Entry point #2--set the time

01F4                SETCLK EQU *
01F4 BDB3ED                JSR $B3ED        get descriptor addr
01F7 1F02                  TFR D,Y          need A & B, so use Y
01F9 A6A4                  LDA ,Y           get length
01FB 8106                  CMPA #6          valid length?
01FD 2620                  BNE SC02         no, skip
01FF 10AE22                LDY 2,Y          get string address
0202 308C46                LEAX <HOURS,PCR  point to parameter list
0205 C603                  LDB #3           there are 3 variables
0207 3404                  PSHS B           save count

0209                SC01   EQU *
0209 ECA1                  LDD ,Y++         get 2 characters
020B 8D13                  BSR DECBIN       convert to binary
020D A780                  STA ,X+          save variable
020F 6AE4                  DEC ,S           done all 3?
0211 26F6                  BNE SC01         loop til done
0213 3504                  PULS B           cleanup stack
0215 6F84                  CLR ,X           reset cycle count
0217 A61D                  LDA -3,X         get hours
0219 8118                  CMPA #24         valid?
021B 2502                  BLO SC02         ok, skip
021D 6F1D                  CLR -3,X         use 0 instead

021F                SC02   EQU *
021F 39                    RTS              return

0220                DECBIN EQU *
0220 8030                  SUBA #$30        strip off ASCII
0222 C030                  SUBB #$30        for both digits
0224 3404                  PSHS B           save LSDigit
0226 C60A                  LDB #10
0228 3D                    MUL              MSDigit times 10
0229 1F98                  TFR B,A          initialize number
022B E6E0                  LDB ,S+          get LSD, set flags

022D                DB01   EQU *
022D 27F0                  BEQ SC02         LSD = 0, done
022F 4C                    INCA             bump the number
0230 5A                    DECB             countdown LSD
0231 20FA                  BRA DB01         loop til done

0233                DB02   EQU *
0233 813C                  CMPA #60         valid number?
0235 2501                  BLO DB03         yes, skip
0237 4F                    CLRA             no, use 0
```

```
0238                    DB03    EQU *
0238 39                         RTS             return to caller

                        * Entry point #3--set screen display position

0239 0400               SCRPOS  FDB $0400       u.l. corner default

023B                    DSPPOS  EQU *
023B BDB3ED                     JSR $B3ED       get "PRINT @" value
023E 108301F8                   CMPD #504       bottom of screen?
0242 2206                       BHI DP01        yes, forget it
0244 C30400                     ADDD #$400      adjust to true address
0247 ED8CEF                     STD SCRPOS,PCR  and save the result

024A                    DP01    EQU *
024A 39                         RTS             return

                        * Interrupt Servicing Routine

024B 00                 HOURS   FCB $00
024C 00                 MINS    FCB $00
024D 00                 SECS    FCB $00
024E 00                 CYCLES  FCB $00

024F                    CLOCK   EQU *
024F 308CF9                     LEAX HOURS,PCR  point to parameter list
0252 EE8CE4                     LDU SCRPOS,PCR  get print position
0255 6C03                       INC 3,X         bump cycles
0257 A603                       LDA 3,X
0259 813C                       CMPA #60        1 second yet?
025B 2520                       BLO CLK01       no, leave
025D 6F03                       CLR 3,X         reset cycles
025F 6C02                       INC 2,X         bump seconds
0261 A602                       LDA 2,X
0263 813C                       CMPA #60        1 minute yet?
0265 2516                       BLO CLK01       no, leave
0267 6F02                       CLR 2,X         reset seconds
0269 6C01                       INC 1,X         bump minutes
026B A601                       LDA 1,X
026D 813C                       CMPA #60        1 hour yet?
026F 250C                       BLO CLK01       no, leave
0271 6F01                       CLR 1,X         reset minutes
0273 6C84                       INC ,X          bump hours
0275 A684                       LDA ,X
0277 8118                       CMPA #24        1 day yet?
0279 2502                       BLO CLK01
027B 6F84                       CLR ,X          reset hours
```

```
027D                    CLKO1   EQU *
027D 6D8DFF72                   TST FLAG,PCR    display mode on?
0281 271A                       BEQ CLKDUN      no, leave
0283 6D03                       TST 3,X         cycles = 0?
0285 2616                       BNE CLKDUN      not yet, leave
0287 C603                       LDB #3          display 3 variables
0289 3404                       PSHS B          save count
028B 2004                       BRA CLKO3       skip the colon

028D                    CLKO2   EQU *
028D 863A                       LDA #':         get a colon
028F A7C0                       STA ,U+         on screen

0291                    CLKO3   EQU *
0291 E680                       LDB ,X+         get current variable
0293 8D0B                       BSR BINDEC      convert to decimal
0295 EDC1                       STD ,U++        both chars on screen
0297 6AE4                       DEC ,S          count down
0299 26F2                       BNE CLKO2       loop til done
029B 3504                       PULS B          cleanup stack

029D                    CLKDUN  EQU *
029D 7E                         FCB $7E         "JMP" opcode

029E 0000               PATCH   FDB $0000       where to go when done

02A0                    BINDEC  EQU *
02A0 4F                         CLRA            set MSD to 0

02A1                    BD01    EQU *
02A1 C00A                       SUBB #10        subtract 10 from number
02A3 2B03                       BMI BD02        too far, leave
02A5 4C                         INCA            bump MSD
02A6 20F9                       BRA BD01        loop til done

02A8                    BD02    EQU *
02A8 8B30                       ADDA #$30       make ASCII
02AA CB3A                       ADDB #$3A       make ASCII; fix LSD
02AC 39                         RTS             return to caller

                        * Entry point #4--enable clock

02AD                    ENABLE  EQU *
02AD 3401                       PSHS CC         save BASIC's intrpt status
02AF 1A50                       ORCC #$50       disable IRQ & FIRQ for now
02B1 308C9B                     LEAX CLOCK,PCR  get our interrupt addr
02B4 BC010D                     CMPX $10D       already setup?
02B7 2709                       BEQ EN01        yes, leave
02B9 FE010D                     LDU $10D        get BASIC's vector
```

```
02BC EF8CDF             STU PATCH,PCR   save it for jump
02BF BF010D             STX $10D        save our vector

02C2            EN01    EQU *
02C2 3581               PULS CC,PC      restore & return

                * Entry point #5--disable clock

02C4            DISABL  EQU *
02C4 3401               PSHS CC         save interrupt status
02C6 1A50               ORCC #$50       disable IRQ & FIRQ
02C8 EC8CD3             LDD PATCH,PCR   get vector
02CB 2703               BEQ DA01        undefined, leave
02CD FD010D             STD $10D        restore for BASIC

02D0            DA01    EQU *
02D0 3581               PULS CC,PC      restore & return

02D2                    END ONOFF

NO ERRORS FOUND
```

Entry Requirements--CLOCK INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable OC--this variable MUST remain properly defined as long as the clock routine is in memory.

2. The INIT subroutine must be called at least once before any of the other subroutines can be called.

3. Note that the INIT routine automatically falls through to the ENABLE CLOCK routine, so you don't have to explicitly call the ENABLE routine following the INIT call.

Exit Conditions--CLOCK INIT

1. INIT defines USR 0, USR 1, and USR 2 for use by the other subroutines.

2. All variables except OC are released for re-use.

Entry Requirements--ENABLE CLOCK

None.

Exit Conditions--ENABLE CLOCK

None.

Entry Requirements--DISABLE CLOCK

None.

Exit Conditions--DISABLE CLOCK

1. The clock is physically removed from the interrupt service routine. If, later on in your program, you wish to re-use the clock, you can do so by making a call to the ENABLE CLOCK routine, which will put the routine back into the interrupt handler.

Entry Requirements--DISPLAY ON/OFF

1. To allow the time to be displayed, simply define variable O1$ = "O" (that's an upper case 'oh', not a zero.) This will cause the time to be displayed in the last defined display position. The time is displayed in inverse video in the format "HH:MM:SS".

2. To disable the display, define the variable O1$ to be any single character except the letter "O".

Exit Conditions--DISPLAY ON/OFF

1. The display is affected according to the contents of variable O1$.

2. If O1$ is not exactly one character long, nothing will happen; that is, the display will remain in its current state.

Entry Requirements--SET TIME

1. Variable O1$ must be defined as a six-character string of digits in the format "HHMMSS" (see the sample call).

Exit Conditions--SET TIME

None.

Entry Requirements--SET DISPLAY POSITION

1. Variable O1 must contain a value between 0 and 504, corresponding to a valid PRINT @ position on the screen.

The upper limit of 504 is necessary because of the length of the string that is actually printed on the screen. If you do not explicitly define the print position, the default value of zero will be used, causing the time to be displayed in the upper left corner of the screen.

Exit Conditions--SET DISPLAY POSITION

None.

Variables Used

OC, O1, O1$, O2$, O3$, O4$.

Sample Call

```
00010  OC = &H01DD                          * start address
       :GOSUB 51100                         * go initialize
       :O1 = 24                             * top right corner
       :GOSUB 51600                         * set display pos'n
       :O1$ = "154100"                      * dummy time
       :GOSUB 51500                         * go set the clock
       :O1$ = "0"                           * <0>n flag
       :GOSUB 51400                         * turn on display
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listings

```
51099  REM clock init
51100  IF PEEK (OC) = &HBD AND PEEK (OC + 122) = &HA6 THEN
       RETURN
51110  O1$ = "BDB3ED1F01A6844A260BA69802814F27014FA78C0139
       00BDB3ED1F02A6A48106262010AE22308C46C6033404ECA18D13
       A7806AE426F635046F84A61D811825026F1D398030C0303404
       C60A3D1F98E6E027F0
51120  O2$ = "4C5A20FA813C25014F390400BDB3ED108301F82206
       C30400ED8CEF3900000000308CF9EE8CE46C03A603813C2520
       6F036C02A602813C25166F026C01A601813C250C6F016C84A684
       811825026F846D8DFF72
51130  O3$ = "271A6D032616C60334042004863AA7C0E6808D0BEDC16
       AE426F235047E00004FC00A2B034C20F98B30CB3A3934011A50
       308C9BBC010D2709FE010DEF8CDFBF010D358134011A50EC8CD3
       2703FD010D358100
```

```
51140  FOR O1 = 0 TO 81
      :O4$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :O5$ = "&H" + MID$ (O2$, 2 * O1 + 1, 2)
      :O6$ = "&H" + MID$ (O3$, 2 * O1 + 1, 2)
      :POKE OC + O1, VAL (O4$)
      :POKE OC + O1 + 82, VAL (O5$)
      :POKE OC + O1 + 164, VAL (O6$)
      :NEXT
      :DEF USR 0 = OC
      :DEF USR 1 = OC + 23
      :DEF USR 2 = OC + 94

51199  REM  enable clock
51200  EXEC OC + 208
      :RETURN

51299  REM disable clock
51300  EXEC OC + 231
      :RETURN

51399  REM display on/off
51400  O1$ = O1$ + ""
      :O1 = USR 0 (VARPTR (O1$))
      :RETURN

51499  REM set time
51500  O1$ = O1$ + ""
      :O1 = USR 1 (VARPTR (O1$))
      :RETURN
51599  REM set display position
51600  O1 = USR 2 (O1)
      :RETURN
```

INVERT INIT (GOSUB 51700)
INVERT MAIN (GOSUB 51800)

Purpose

Sometimes, a BASIC program requires a short routine to cause certain characters to be displayed on the screen in inverse video. When the characters to be displayed are letters of the alphabet, the process is accomplished by a normal PRINT statement, but when you want to display a non-letter character in inverse video, you must resort to the POKE statement. The POKE command is fine if you are only displaying one or two characters on the screen, but any more than that, and you begin to notice how slow the command really is; not only that, but for each POKE that you execute, you must determine the proper address on the screen.

The INVERT routine eliminates the headaches that can arise from the mundane task of calculating (by trial and error) numerous screen addresses. It takes a normal string as input and displays each character in inverse video, starting at the current print position. The only restriction is that the string must contain characters whose ASCII values are in the range &H20 to &H5F (32 to 95). Characters outside this range will be printed, but the results will not be what you expect.

The string itself is retained, and can be re-used for other purposes if desired; only the display is affected. INVERT does not do any scrolling; instead, it simply stops printing if the length of the passed string will cause characters to be displayed beyond the lower right corner of the screen.

As an example, suppose you wanted to display the message "<<< ANY KEY TO CONTINUE >>>" in inverse video at the bottom of the screen. All you have to do is execute a dummy PRINT @ statement to position the cursor and then pass the string to the subroutine. As you'll quickly discover, INVERT will do the job much faster than you could do it with POKE commands.

Assembly Language Listing

```
                        ********************************
                        *   INVERSE VIDEO GENERATION   *
                        ********************************

0E00                          ORG $01DD       in cassette buffer

01DD                    INVERT EQU *
01DD BDB3ED                   JSR $B3ED       get descriptor addr
```

```
01E0 1F02              TFR D,Y         need A & B, use Y
01E2 9E88              LDX $88         get screen address
01E4 E6A4              LDB ,Y          get length
01E6 2719              BEQ IN03        null string, leave
01E8 10AE22            LDY 2,Y         point to actual string

01EB             IN01  EQU *
01EB 8C05FF            CMPX #$05FF     end of screen?
01EE 2211              BHI IN03        yes, leave
01F0 A6A0              LDA ,Y+         get a char
01F2 817F              CMPA #$7F       graphic?
01F4 2206              BHI IN02        yes, skip
01F6 8140              CMPA #$40       normal char?
01F8 2502              BLO IN02        already inverted, skip
01FA 8040              SUBA #$40       invert char

01FC             IN02  EQU *
01FC A780              STA ,X+         on screen
01FE 5A                DECB            countdown length
01FF 26EA              BNE IN01        loop til done

0201             IN03  EQU *
0201 39                RTS             done, leave

0202                   END INVERT

NO ERRORS FOUND
```

Entry Requirements--INVERT INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--INVERT INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--INVERT MAIN

1. The string to be displayed in inverse video must be passed in variable 01$.

Exit Conditions--INVERT MAIN

1. The string is displayed starting at the current cursor location.

2. BASIC's cursor pointer is unchanged; that is, subsequent unlocated PRINTs will overwrite the data just displayed.

3. Variable O1$ is unchanged and may be re-used.

Variables Used

O0, O1, O1$, O2$

Sample Call

```
00010  CLS
       :O0 = &H01DD                          * where to put it
       :GOSUB 51700                          * go initialize
       :O1$ = " !" + CHR$ (34) + "#$%&'()*+,-./0123456789:;<=>?
        @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]" + CHR$ (&H5F)
00020  PRINT O1$                             * display original
       :PRINT @224,;                         * set print pos'n
       :GOSUB 51800                          * go display inverse
       :PRINT @416,;                         * another dummy print
       :FOR I = 1 TO 1000
       :NEXT
00030  CLS
       :O1$ = "GREETINGS!!!"
00040  PRINT @234,O1$                        * flash a message
       :PRINT @234,;
       :FOR I = 1 TO 50
       :NEXT
       :GOSUB 51800
       :FOR I = 1 TO 50
       :NEXT
       :GOTO 40                              * until BREAK pressed
```

Subroutine Listings

```
51699  REM invert init
51700  IF PEEK (O0) = &HBD AND PEEK (O0 + 36) = &H39 THEN
       RETURN
       :ELSE O1$ = "BDB3ED1F029E88E6A4271910AE228C05FF2211
        A6A0817F22068140250280407805A26EA39"
```

```
51710  FOR O1 = 0 TO 36
      :O2$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :POKE OO + O1, VAL (O2$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN

51799  REM invert main
51800  O1$ = O1$ + ""
      :O1 = USR 0 (VARPTR (O1$))
      :RETURN
```

SCRAMB INIT (GOSUB 51900)
SCRAMB MAIN (GOSUB 52000)

Purpose

If you have ever wanted to hide sensitive data from that person who just dropped in, without shutting down the computer, then you probably have a need for a subroutine such as SCRAMB. This is a very short and simple routine that does nothing more than scramble the video screen. A second call to the same routine causes the screen to be restored to its original contents.

Suppose, for example, that you are writing a data-base management program which you ultimately hope to sell. You could incorporate SCRAMB into your program in such a way that, whenever the user wanted to hide the screen data (during data entry or editing), all he would have to do is press a single pre-defined key. Then your program would go into a special subroutine which would immediately call SCRAMB, and then wait for a selection of keystrokes (a password, if you will) that the user might have defined during a configuration phase.

The logic that is used in the routine could easily be applied to a larger, high-resolution display. We'll leave this modification to you....

Assembly Language Listing

```
                    ********************************
                    *       SCREEN  SCRAMBLE       *
                    ********************************

0E00                        ORG $01DD        in cassette buffer

01DD                SCRAMB  EQU *
01DD 8E0400                 LDX #$0400       get screen start addr

01E0                SCR01   EQU *
01E0 6080                   NEG ,X+          one byte at a time
01E2 8C0600                 CMPX  #$0600     end of screen?
01E5 25F9                   BLO SCR01        loop til done
01E7 39                     RTS              done, leave

01E8                        END SCRAMB

NO ERRORS FOUND
```

Entry Requirements--SCRAMB INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable OS, which must remain defined as long as the routine is in memory.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--SCRAMB INIT

1. All variables, except OS, are released for re-use.

Entry Requirements--SCRAMB MAIN

None.

Exit Conditions--SCRAMB MAIN

1. The contents of each screen location are negated. The screen can be completely restored by simply making a second call to the same routine.

Variables Used

OS, O1, O1$, O2$

Sample Call

```
00010  OS = &H01DD                          * where to put it
      :GOSUB 51900                          * go initialize
00020  A$ = INKEY$                          * infinite loop
      :IF A$ = "" THEN 20
      :ELSE GOSUB 52000                     * scramble/restore
      :GOTO 20
```

Subroutine Listings

```
51899  REM scramb init
51900  IF PEEK (OS) = &H8E AND PEEK (OS + 10) = &H39 THEN
       RETURN
      :ELSE O1$ = "8E040060808C060025F939"
      :FOR O1 = 0 TO 10
      :O2$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :POKE OS + O1, VAL (O2$)
      :NEXT
      :RETURN
```

```
51999  REM scramb main
52000  EXEC OS
      :RETURN
```

BORDER INIT (GOSUB 52100)
BORDER MAIN (GOSUB 52200)

Purpose

This routine allows you to enhance any screen display by placing a border around the screen's perimeter. The border consists of repeated copies of a single ASCII value between &H00 and &HFF (0 to 255). In the sample call, all possible values are sent to the routine. Note the speed of execution.

Assembly Language Listing

```
                    ********************************
                    *       BORDER GENERATOR       *
                    ********************************

0E00                        ORG $01DD       in cassette buffer

01DD                BORDER  EQU *
01DD BDB3ED                 JSR $B3ED       get border character
01E0 1F98                   TFR B,A         make a double copy
01E2 8E0400                 LDX #$0400      u.l. corner

01E5                BD01    EQU *
01E5 ED8901E0               STD $1E0,X      do bottom line
01E9 ED81                   STD ,X++        and top line
01EB 8C0420                 CMPX #$420      at 2nd line yet?
01EE 26F5                   BNE BD01        loop til done
01F0 C60E                   LDB #14         14 lines to go

01F2                BD02    EQU *
01F2 A784                   STA ,X          left side
01F4 A7881F                 STA 31,X        and right side
01F7 308820                 LEAX 32,X       next line
01FA 5A                     DECB            done all lines?
01FB 26F5                   BNE BD02        loop til don
01FD 39                     RTS             and return

01FE                        END BORDER

NO ERRORS FOUND
```

Entry Requirements--BORDER INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable OO.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--BORDER INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--BORDER MAIN

1. The border value must be passed in variable 01.

Exit Conditions--BORDER MAIN

None.

Variables Used

00, 01, 01$, 02$

Sample Call

```
00010  CLS
       :00 = &H01DD                    * where to put it
       :GOSUB 52100                    * go initialize
       :FOR I = 0 TO 255               * all possible borders
       :01 = I
       :GOSUB 52200                    * draw border
       :NEXT
       .
       .                               * more instructions
       .
30000  END
```

Subroutine Listings

```
52099  REM border init
52100  IF PEEK (00) = &HBD AND PEEK (00 + 32) = &H39 THEN
       RETURN
       :ELSE 01$ = "BDB3ED1F988E0400ED8901E0ED818C042026F5
       C60EA784A7881F3088205A26F539"
       :FOR 01 = 0 TO 32
       :02$ = "&H" + MID$ (01$, 2 * 01 + 1, 2)
       :POKE 00 + 01, VAL (02$)
       :NEXT
       :DEF USR 0 = 00
       :RETURN
```

```
52199  REM border main
52200  O1 = USR 0 (O1)
      :RETURN
```

MLSET INIT (GOSUB 52300)
MLSET MAIN (GOSUB 53400)

Purpose

This routine is specifically designed to speed up the process of entering machine language code (or graphic screen data) into memory and is particularly useful when your program contains a large amount of data which would normally take a fairly long time to POKE.

Suppose, for example, that you want to include CLOCK, BORDER, FLASH, and INVERT in your program. Obviously, these routines involve a fair amount of data to POKE into memory. The solution is to POKE just the MLSET routine into some unused area of memory. Then you must modify the four routines so they call MLSET instead of doing the POKEing themselves. These modifications should include ensuring that each routine has a different USR function number. Now, when you RUN the program, you will notice a drastic increase in speed. Note that once the last block of data is POKEd into memory, the memory occupied by MLSET may be freed for use by some other part of your program.

MLSET accepts a character string of hexadecimal digits as input, converts the data to binary, and stores the results in a contiguous block of memory, whose starting address may be specified within the string. If no address is specified, MLSET uses a default starting address of &H0400 (1024), which is the upper left corner of the screen. (Hopefully, this address will prevent the potentially fatal POKEing of data into the wrong place!) When you make multiple calls to MLSET without specifying addresses in the second and subsequent calls, MLSET will store data immediately following the last byte of the previous block. Thus, in the example above, if you specified a start address only for CLOCK, the other routines would immediately follow CLOCK in memory.

The format for the input string is pretty restrictive: it must contain an even number of hexadecimal digits (upper case letters from A to F and/or numeric digits from 0 to 9), and it may contain any number of individual "@" symbols, each of which must precede a 4-digit memory address, as in the following example:

```
O1$ = "@0400FFFFFFFF@041CFFFFFFFF"
```

This example would cause four orange graphic blocks to appear in the upper left corner and upper right corner of the screen.

MLSET does not perform any error checking on the input string except to verify that it is not a null string. If you place invalid characters in the string, the routine will continue to function correctly, but obviously the results will be completely unpredictable and definitely untrustworthy.

Assembly Language Listing

```
                        ********************************
                        *      M.L. SETUP ROUTINE      *
                        ********************************

0E00                            ORG $01DD       in cassette buffer

01DD                    MLSET   EQU *
01DD BDB3ED                     JSR $B3ED       get descriptor addr
01E0 1F01                       TFR D,X         need A & B, so use X
01E2 EE8C2D                     LDU <ADDR,PCR   get destination addr
01E5 E684                       LDB ,X          get string length
01E7 3404                       PSHS B          save length
01E9 2722                       BEQ MLDUN       length = 0, leave
01EB AE02                       LDX 2,X         get string address

01ED                    MLO1    EQU *
01ED A680                       LDA ,X+         get next char
01EF 8140                       CMPA #'@        flag for new address?
01F1 2621                       BNE MLO2        no, skip
01F3 EC81                       LDD ,X++        get 2 hex nybbles
01F5 8D2D                       BSR HEXBIN      go convert to binary
01F7 3402                       PSHS A          save MSB
01F9 EC81                       LDD ,X++        get 2 more
01FB 8D27                       BSR HEXBIN      convert
01FD 1F89                       TFR A,B         get LSB
01FF 3502                       PULS A          get MSB
0201 1F03                       TFR D,U         update address pointer
0203 E6E4                       LDB ,S          get length
0205 C005                       SUBB #5         have used up 5 chars
0207 E7E4                       STB ,S          replace length
0209 2502                       BCS MLDUN       overflow, leave
020B 26E0                       BNE MLO1        loop til done

020D                    MLDUN   EQU *
020D EF8C02                     STU <ADDR,PCR   save new destination
0210 3584                       PULS B,PC       cleanup & leave

0212 0400               ADDR    FDB $0400       default destination
```

```
0214                ML02    EQU *
0214 6AE4                   DEC ,S          count down length
0216 27F5                   BEQ MLDUN       done, leave
0218 E680                   LDB ,X+         get LS nybble of byte
021A 8D08                   BSR HEXBIN      convert to binary
021C A7C0                   STA ,U+         store binary in memory
021E 6AE4                   DEC ,S          count down length
0220 27EB                   BEQ MLDUN       done, leave
0222 20C9                   BRA ML01        loop til done

0224                HEXBIN  EQU *
0224 8030                   SUBA #$30       remove ASCII
0226 C030                   SUBB #$30       from both nybbles
0228 8109                   CMPA #9         numeric digit?
022A 2302                   BLS HB01        ok, skip
022C 8007                   SUBA #7         adjust for A to F

022E                HB01    EQU *
022E C109                   CMPB #9         repeat for LS nybble
0230 2302                   BLS HB02
0232 C007                   SUBB #7

0234                HB02    EQU *
0234 48                     ASLA            MS nybble to far left
0235 48                     ASLA
0236 48                     ASLA
0237 48                     ASLA
0238 C40F                   ANDB #$0F       keep bits 0-3
023A 3404                   PSHS B          temp save
023C AAE0                   ORA ,S+         assemble one byte
023E 39                     RTS             return to caller

023F                        END MLSET

NO ERRORS FOUND
```

Entry Requirements--MLSET INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--MLSET INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--MLSET MAIN

1. The string containing the data to be POKEd must be passed in variable O1$.

Exit Conditions--MLSET MAIN

1. The data is stored in memory starting at the specified address. If no address is specified, the data is stored immediately following the last byte of the previously sent string. The initial default address is &H0400.

2. Variable O1$ is unchanged and may be re-used.

Variables Used

O0, O1, O1$, O2$

Sample Call

```
00010  CLS
       :O1 = &H01DD                          * where to put it
       :GOSUB 52300                          * go initialize
       :O1$ = "@0520FFFFFEFEFDFDFCFCFBFBFAFA"
       :GOSUB 52400                          * display screen data
       .
       .                                     * more instructions
       .
30000  END
```

Subroutine Listings

```
52299  REM mlset init
52300  IF PEEK (OO) = &HBD AND PEEK (OO + 97 = &H39 THEN
       RETURN
52310  O1$ = "BDB3ED1F01EE8C2DE68434042722AE02A68081402621
       EC818D2D3402EC818D271F8935021F03E6E4C005E7E4250226E0
       EF8C02358404006AE427F5E6808D08A7C06AE427EB20C98030
       C030810923028007C1092302C00748484848C40F3404AAE039"
52320  FOR O1 = 0 TO 97
      :O2$ = "&H" + MID$ (O1$, O1 * 2 + 1, 2)
      :POKE (OO + O1), VAL (O2$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN
52399  REM mlset main
52400  O1$ = O1$ + ""
      :O1 = USR 0 (VARPTR (O1$))
      :RETURN
```

RESTORE INIT (GOSUB 52500)
RESTORE MAIN (GOSUB 52600)

Purpose

If you have ever written a BASIC program containing numerous DATA statements, you will undoubtedly have noticed that it is extremely inconvenient to try to position the data pointer to a specific set of values somewhere in the middle of a data list. The operation requires that you RESTORE to the beginning of your program and perform enough dummy READs to point to the desired data item. This procedure is both cumbersome and time-consuming.

The RESTORE routine included here is designed to alleviate this problem by allowing you to position the data pointer to the beginning of any BASIC program line. All you have to do is preset a variable to the desired line number and call the routine. If you pass a line number which does not exist, RESTORE will move the data pointer to the beginning of the program.

One thing you should be aware of is that when you re-number your program, you will have to manually make changes to those lines containing calls to the routine. If you don't, RESTORE will move the pointer to the wrong place, and, consequently, your program will produce unexpected results.

Assembly Language Listing

```
                    ********************************
                    *   RESTORE TO SPECIFIC LINE   *
                    ********************************

0E00                        ORG $01DD       in cassette buffer

01DD                RESTOR  EQU *
01DD BDB3ED                 JSR $B3ED       get the line number
01E0 9E19                   LDX $19         get BASIC start addr

01E2                RST01   EQU *
01E2 10A302                 CMPD 2,X        line number found?
01E5 2708                   BEQ RST02       yes, leave
01E7 AE84                   LDX ,X          get link to next line
01E9 911B                   CMPA $1B        end of BASIC program?
01EB 25F5                   BLO RST01       loop til end
01ED 9E19                   LDX $19         use 1st line instead
```

```
01EF              RST02  EQU *
01EF 301F                LEAX -1,X       1 byte ahead of line
01F1 9F33                STX $33         update data pointer
01F3 39                  RTS             back to BASIC

01F4                     END RESTOR

NO ERRORS FOUND
```

Entry Requirements--RESTORE INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--RESTORE INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--RESTORE MAIN

1. The desired line number you wish to RESTORE to must be passed in variable 01. The line number must be in the range 0 to 63999. If the line number is outside this range, or if it does not exist, RESTORE will move the pointer to the beginning of the program.

Exit Conditions--RESTORE MAIN

1. If the line number is outside the allowable range, or if the line number does not exist, RESTORE will move the pointer to the beginning of the program; otherwise it will be moved to the specified line.

Variables Used

00, 01, 01$, 02$

Sample Call

```
00010  00 = &H01DD                    * where to put it
       :GOSUB 52500                   * go initialize
       :DATA 10,20,30                 * some dummy data
00020  DATA 11,22,33                  * data to read
```

```
00030  O1 = 20                              * RESTORE to line 20
      :GOSUB 52600                          * go do it
00040  READ A                               * and prove it!
      :PRINT A
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listings

```
52499  REM restore init
52500  IF PEEK (OO) = &HBD AND PEEK (OO + 22) = &H39 THEN
       RETURN
      :ELSE O1$ = "BDB3ED9E1910A3022708AE849C1B25F59E19
       301F9F3339"
      :FOR O1 = 0 TO 22
      :O2$ = "&H" + MID$ (O1$, O1 * 2 + 1, 2)
      :POKE (OO + O1), VAL (O2$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN

52599  REM restore main
52600  IF O1 < 0 OR O1 > 65399 THEN RESTORE
      :RETURN
      :ELSE O1 = USR 0 (O1 + (O1 > 32767) * 65536)
      :RETURN
```

PACK INIT (GOSUB 52700)
PACK MAIN (GOSUB 52800)

Purpose

This unique subroutine allows you to take an 8-character date string of the form "DD/MM/YY" and compress it into two bytes. The same routine will also convert the packed 2-byte string back into its original unpacked format.

The routine is particularly useful in a BASIC program which involves external files that will contain a lot of date information (e.g. a check register program or a stock management program). Obviously, if the dates are packed before being inserted into the file, this can result in a significant saving in space over a large file.

Assembly Language Listing

```
                        ********************************
                        *   PACK/UNPACK DATE STRING    *
                        ********************************

0E00                            ORG $01DD       in cassette buffer

01DD                    PACK    EQU *
01DD BDB3ED                     JSR $B3ED       get descriptor addr
01E0 1F02                       TFR D,Y         need A & B, so use Y
01E2 E6A4                       LDB ,Y          get length
01E4 C102                       CMPB #2         compressed string?
01E6 2744                       BEQ UNPACK      yes, go de-compress
01E8 C108                       CMPB #8         valid unpacked string?
01EA 2625                       BNE PK01        no, do nothing, leave
01EC EE22                       LDU 2,Y         get string address
01EE ECC4                       LDD ,U          get 1st 2 bytes (day)
01F0 8D20                       BSR DECBIN      convert to binary
01F2 58                         ASLB            move bits into position
01F3 58                         ASLB
01F4 58                         ASLB
01F5 3404                       PSHS B          save day mask
01F7 EC43                       LDD 3,U         next 2 bytes (month)
01F9 8D17                       BSR DECBIN      convert to binary
01FB 3404                       PSHS B          just save it for now
01FD EC46                       LDD 6,U         next 2 bytes (year)
01FF 8D11                       BSR DECBIN      convert to binary
0201 58                         ASLB            remove MS bit
0202 66E4                       ROR ,S          LS month bit to carry
0204 56                         RORB            and into year byte
0205 3502                       PULS A          get month
```

```
0207 8407              ANDA #7          clear unneeded bits
0209 AAE0              ORA ,S+          mask with day byte
020B EDC4              STD ,U           put data in strg area
020D C602              LDB #2           new length = 2
020F E7A4              STB ,Y           save new length

0211            PK01   EQU *
0211 39                RTS              back to BASIC

0212            DECBIN EQU *
0212 8030              SUBA #$30        strip ASCII from MSD
0214 2B0C              BMI DCERR        invalid char, leave
0216 C030              SUBB #$30        and LSD
0218 2B08              BMI DCERR
021A 3404              PSHS B           save LSD
021C C60A              LDB #10          multiply MSD by 10
021E 3D                MUL
021F EBE0              ADDB ,S+         add in LSD
0221 21                FCB $21          "BRN"--skip 1 byte

0222            DCERR  EQU *
0222 5F                CLRB             error; force zero
0223 39                RTS

0224            STRG   RMB 8            area for unpacked strg

022C            UNPACK EQU *
022C 338CF5            LEAU STRG,PCR    get new string address
022F ECB802            LDD [2,Y]        get compressed data
0232 EF22              STU 2,Y          update string address
0234 A7C4              STA ,U           temp save
0236 1F98              TFR B,A          get year data in A & B
0238 847F              ANDA #$7F        now have proper year
023A 3402              PSHS A           save year
023C C480              ANDB #$80        remove unneeded bits
023E 3404              PSHS B           save bit for month
0240 A6C4              LDA ,U           get day/month data
0242 1F89              TFR A,B          two copies
0244 44                LSRA             reposition bits--
0245 44                LSRA             puts day in A
0246 44                LSRA
0247 C40F              ANDB #$0F        remove unneeded bits
0249 69E0              ROL ,S+          year bit into carry
024B 59                ROLB             and into month
024C 3404              PSHS B           save month
024E 8D18              BSR BINDEC       convert day to decimal
0250 EDC4              STD ,U           save day in string
0252 3502              PULS A           get month
0254 8D12              BSR BINDEC       convert to decimal
```

```
0256 ED43              STD 3,U          save month in string
0258 3502              PULS A           get year
025A 8D0C              BSR BINDEC       convert to decimal
025C ED46              STD 6,U          save year in string
025E CC2F08            LDD #$2F08       A = "/"; B = new length
0261 A742              STA 2,U          save 2 slashes
0263 A745              STA 5,U
0265 E7A4              STB ,Y           save length
0267 39                RTS              return to BASIC

0268            BINDEC EQU *
0268 5F                CLRB             MSD = 0

0269            BD01   EQU *
0269 800A              SUBA #10         subtract to find MSD
026B 2503              BCS BD02         too many subs, leave
026D 5C                INCB             bump MSD
026E 20F9              BRA BD01         loop for whole number

0270            BD02   EQU *
0270 8B3A              ADDA #$3A        adjust & make ASCII
0272 CB30              ADDB #$30        make ASCII
0274 1E89              EXG A,B          digits in proper order
0276 39                RTS

0277                   END PACK

NO ERRORS FOUND
```

Entry Requirements--PACK INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--PACK INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--PACK MAIN

1. The string to be packed/unpacked must be passed in variable 01$.

Exit Conditions--PACK MAIN

1. The string will be packed if it is exactly eight characters in length and unpacked if it is exactly two characters in length. The string will be unaffected if it is any other length.

2. The routine does not perform any error checking on the contents of the string. It is up to you to ensure that the string has the proper format.

Variables Used

OO, O1, O1$, O2$, O3$, O4$.

Sample Call

```
00010  OO = &HO1DD                          * where to put it
       :GOSUB 52700                         * go initialize
       :O1$ = "10/10/83"                    * test string
       :GOSUB 52800                         * go pack
       :PRINT O1$                           * display result
00020  GOSUB 52800                          * go unpack
       :PRINT O1$                           * display result
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listings

```
52699  REM pack init
52700  IF PEEK (OO) = &HBD AND PEEK (OO + 153) = &H39 THEN
       RETURN
52710  O1$ = "BDB3ED1F02E6A4C1022744C1082625EE22ECC48D20
       5858583404EC438D173404EC468D115866E45635028407AAE0
       EDC4C602E7A43980302B0CC0302B083404C60A3DEBE0215F39
       000000000000"
52720  O2$ = "0000338CF5ECB802EF22A7C41F98847F3402C4803404
       A6C41F89444444C40769E05934048D18EDC435028D12ED433502
       8D0CED46CC2F08A742A745E7A4395F800A25035C20F98B3A
       CB301E8939"
```

```
52730  FOR O1 = 0 TO 76
       :O3$ = "&H" + MID$ (O1$, O1 * 2 + 1, 2)
       :O4$ = "&H" + MID$ (O2$, O1 * 2 + 1, 2)
       :POKE (OO + O1), VAL (O3$)
       :POKE (OO + O1 + 77), VAL (O4$)
       :NEXT
       :DEF USR 0 = OO
       :RETURN

52799  REM pack main
52800  O1$ = O1$ + ""
       :O1 = USR 0 (VARPTR (O1$))
       :O1$ = O1$ + ""
       :RETURN
```

EDLIN INIT (GOSUB 52900)
EDLIN MAIN (GOSUB 53000)

Purpose

This rather lengthy subroutine allows you to input and/or edit a fixed-length string. It is a little unusual in that it uses the video screen as a buffer. For this reason, the string must be displayed on the screen and the cursor re-positioned to the start of the string before the routine is called.

EDLIN allows the use of the left and right arrow keys for positioning the cursor over a specific character in the string. The <BREAK> key causes the character under the cursor to be deleted; <CLEAR> inserts a blank at the current cursor position; and <ENTER> signifies that editing is complete and causes a return to the calling program. All other keys are treated as ordinary keystrokes, which are placed on the screen at the cursor position. Of course, the cursor automatically advances forward each time a non-control key is pressed. All keys are auto-repeating.

When making the call to EDLIN, ensure that a fixed-length string is defined and displayed on the screen at the desired location. For input purposes, the string may consist of a fixed number of spaces; for editing purposes, the string may contain any combination of characters. On return, the string may have been altered with respect to its contents, but its length will remain the same as it was on entry.

Assembly Language Listing

```
                        ********************************
                        *      EDITED LINE INPUT       *
                        ********************************

                        * The following variables are all referenced
                        * relative to the hardware stack

0E00                            ORG $00

0000                    BEGLIN  RMB 2           start of line
0002                    ENDLIN  RMB 2           end of line
0004                    LENGTH  RMB 1           length of line
0004                    DATA    EQU LENGTH      screen data
0005                    CURPOS  RMB 2           temporary value
```

```
0007                    STRING RMB 2            address of string
0009                    REPEAT RMB 1            key repeat flag
000A                    BLINK  RMB 1            cursor blink rate
000B                    NUMVAR EQU *            total number of variables

                        * Constants

001F                    EMARK  EQU $1F          end of line marker
0400                    DELAY1 EQU $0400        pre-repeat delay
0040                    DELAY2 EQU $0040        repeat delay

                        * Main routine

000B                           ORG $01DD        in cassette buffer

01DD                    EDLIN  EQU *
01DD BDB3ED                    JSR $B3ED        get string descriptor addr
01E0 1F03                      TFR D,U          need A & B, use U
01E2 E6C4                      LDB ,U           get length of string
01E4 2601                      BNE EL01         ok, skip
01E6 39                        RTS              back to BASIC

01E7                    EL01   EQU *
01E7 3275                      LEAS -NUMVAR,S   reserve variable space
01E9 6F69                      CLR REPEAT,S     initialize for use
01EB E764                      STB LENGTH,S     save string length
01ED EC42                      LDD 2,U          get actual string address
01EF ED67                      STD STRING,S     save for later

01F1                    EL02   EQU *
01F1 DC88                      LDD $88          get cursor address
01F3 EDE4                      STD BEGLIN,S     save start point
01F5 EB64                      ADDB LENGTH,S    find end of string
01F7 8900                      ADCA #0
01F9 ED62                      STD ENDLIN,S     save end address
01FB 830600                    SUBD #$0600      off screen?
01FE 2504                      BLO EL03         no, continue
0200 6A64                      DEC LENGTH,S     reduce string length
0202 20ED                      BRA EL02

0204                    EL03
0204 861F                      LDA #EMARK       get end marker
0206 A7F802                    STA [ENDLIN,S]   mark end of line
0209 AEE4                      LDX BEGLIN,S     get screen start
```

```
                    * Flash cursor and wait for keypress

020B                CURSOR EQU *
020B 108E0400              LDY #DELAY1      repeat delay value
020F 6D69                  TST REPEAT,S     is key being pressed?
0211 2704                  BEQ CUR01        no, skip
0213 108E0040              LDY #DELAY2      yes, use this value

0217                CUR01  EQU *
0217 E684                  LDB ,X           get value at cursor
0219 E764                  STB DATA,S       save value at cursor

021B                CUR02  EQU *
021B 6A6A                  DEC BLINK,S      time to flash?
021D 2606                  BNE CUR03
021F E684                  LDB ,X
0221 C840                  EORB #$40        yes, invert character
0223 E784                  STB ,X

0225                CUR03  EQU *
0225 313F                  LEAY -1,Y        decrement repeat counter
0227 260D                  BNE CUR05        if 0, reset rollover table
0229 CCFF08                LDD #$FF08       A=not-pressed flag;B=count
022C CE0152                LDU #$152        start of rollover table

022F                CUR04  EQU *
022F A7C0                  STA ,U+
0231 5A                    DECB
0232 26FB                  BNE CUR04        loop for whole table
0234 A769                  STA REPEAT,S     set repeat flag

0236                CUR05  EQU *
0236 BDA1C1                JSR $A1C1        keypress?
0239 2604                  BNE CUR06        yes, skip
023B 6F69                  CLR REPEAT,S     reset repeat flag
023D 20DC                  BRA CUR02        and loop

023F                CUR06  EQU *
023F C601                  LDB #1           force cursor blink
0241 E76A                  STB BLINK,S
0243 E664                  LDB DATA,S       get original value
0245 E784                  STB ,X           replace it
0247 AF65                  STX CURPOS,S     save cursor position

                    * if ENTER, restore string & return to BASIC

0249                KEYS   EQU *
0249 810D                  CMPA #$0D        <ENTER>--user finished?
024B 261F                  BNE KEY01        no, skip
```

```
024D AEE4              LDX BEGLIN,S     start of screen buffer
024F EE67              LDU STRING,S     start of string

0251            RETURN EQU *
0251 A680              LDA ,X+          get character
0253 2B0E              BMI RET02        store if graphics
0255 8160              CMPA #$60        check for lowercase
0257 2504              BLO RET01        yes, continue
0259 8040              SUBA #$40        no, fix it
025B 2006              BRA RET02        and go store it

025D            RET01  EQU *
025D 8140              CMPA #$40        check for uppercase
025F 2402              BHS RET02        yes, go store
0261 8B60              ADDA #$60        no, fix it

0263            RET02  EQU *
0263 A7C0              STA ,U+          change the string
0265 AC62              CMPX ENDLIN,S    done?
0267 26E8              BNE RETURN       no, loop
0269 326B              LEAS NUMVAR,S    restore stack for BASIC
026B 39                RTS              back to BASIC

                * Determine which function key

026C            KEY01  EQU *
026C 8109              CMPA #9          right arrow?
026E 2604              BNE KEY02        no, skip
0270 3001              LEAX 1,X         bump cursor address
0272 2006              BRA CHECK        go check location

0274            KEY02  EQU *
0274 8108              CMPA #$8         left arrow?
0276 260E              BNE KEY03        no, skip
0278 301F              LEAX -1,X        previous cursor address

027A            CHECK  EQU *
027A ACE4              CMPX BEGLIN,S    before start position
027C 2504              BLO CHK01        yes
027E AC62              CMPX ENDLIN,S    past end?
0280 2502              BLO CHK02        no, continue

0282            CHK01  EQU *
0282 AE65              LDX CURPOS,S     old cursor pos

0284            CHK02  EQU *
0284 2085              BRA CURSOR       go get another keypress
```

```
0286                KEY03   EQU *
0286 8103                   CMPA #$03       <BREAK>--delete char?
0288 2610                   BNE KEY05       no, skip
028A 3001                   LEAX 1,X        point to next position

028C                KEY04   EQU *
028C A680                   LDA ,X+         get character
028E A71E                   STA -2,X        move everything left
0290 AC62                   CMPX ENDLIN,S   finished deleting?
0292 23F8                   BLS KEY04       loop til done
0294 8660                   LDA #$60        put blank at end of line
0296 A71E                   STA -2,X
0298 20E8                   BRA CHK01       loop for more

029A                KEY05   EQU *
029A 810C                   CMPA #$0C       <CLEAR>--insert char?
029C 2612                   BNE KEY07       no, skip
029E AE62                   LDX ENDLIN,S    get end address
02A0 301F                   LEAX -1,X       second last char

02A2                KEY06   EQU *
02A2 A682                   LDA ,-X         get character
02A4 A701                   STA 1,X         move it to the right
02A6 AC65                   CMPX CURPOS,S   done?
02A8 24F8                   BHS KEY06       no, loop
02AA 8660                   LDA #$60        insert a blank
02AC A701                   STA 1,X
02AE 20D2                   BRA CHK01       loop for more

02B0                KEY07   EQU *
02B0 8120                   CMPA #$20       printable?
02B2 25D0                   BLO CHK02       no, leave
02B4 AC62                   CMPX ENDLIN,S   on screen?
02B6 24CC                   BHS CHK02       no, leave
02B8 9F88                   STX $88         save cursor pointer
02BA BDA30A                 JSR $A30A       and go print
02BD 3001                   LEAX 1,X        new cursor
02BF 20C3                   BRA CHK02       loop for more

02C1                        END EDLIN

NO ERRORS FOUND
```

Entry Requirements--EDLIN INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--EDLIN INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--EDLIN MAIN

1. The fixed-length string must be displayed on the screen at the desired location; the cursor must be positioned at the start of the string; and the string itself must be passed to the routine in variable O1$.

Exit Conditions--EDLIN MAIN

None.

Variables Used

O0, O1, O1$, O2$, O3$, O4$.

Sample Call

```
00010  O0 = &H01DD                          * where to put it
       :GOSUB 52900                         * go initialize
       :CLS
       :PRINT @224, "USE LEFT AND RIGHT ARROWS TO    MOVE THE
         CURSOR                    USE <CLEAR> TO INSERT A BLANK
           USE <BREAK> TO DELETE ONE       CHARACTER
00020  O1$ = "THIS IS A TEST STRING"
       :PRINT @0,O1$                        * display string
       :PRINT @0,;                          * reposition cursor
       :GOSUB 53000                         * go do editing
00030  CLS
       :PRINT O1$                           * re-display string
       .
       .                                    * more instructions
       .
30000  END
```

Subroutine Listings

```
52899  REM edlin init
52900  IF PEEK (OO) = &HBD AND PEEK (OO + 227) = &HC3 THEN
       RETURN
52910  O1$ = "BDB3ED1F03E6C426013932756F69E764EC42ED67DC88
       EDE4EB648900ED6283060025046A6420ED861FA7F802AEE4
       108E04006D692704108E0040E684E7646A6A2606E684C840E784
       313F260DCCFF08CE0152A7C05A26FBA769BDA1C126046F6920DC
       C601E76AE664E784AF65810D261FAEE4
52920  O2$ = "EE67A6802B0E8160250480402006814024028B60A7C0
       AC6226E8326B39810926043001200681082 60E301FACE42504
       AC622502AE6520858103261030 01A680A71EAC6223F88660A71E
       20E8810C2612AE62301FA682A701AC6524F88660A70120D28120
       25D0AC6224CC9F88BDA30A300120C3
52930  FOR O1 = 0 TO 113
      :O3$ = "&H" + MID$ (O1$, O1 * 2 + 1, 2)
      :O4$ = "&H" + MID$ (O2$, O1 * 2 + 1, 2)
      :POKE (OO + O1), VAL (O3$)
      :POKE (OO + O1 + 114), VAL (O4$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN

52999  REM edlin main
53000  O1$ = O1$ + ""
      :O1 = USR 0 (VARPTR (O1$))
      :O1$ = O1$ + ""
      :RETURN
```

SEARCH INIT (GOSUB 60000)
SEARCH MAIN (GOSUB 60100)

Purpose

This is a special-purpose subroutine that will be most useful during the design phase of writing your BASIC program. It allows you at any time to perform a search through your program for a specific combination of characters. The routine quickly displays on the screen the number of occurrences of the target string and the line number(s) in which the string was found.

As a specific example, suppose you want to delete a REMark line from your program but you aren't sure if you've made any explicit references to that line (possibly a GOTO or GOSUB). Simply define 01$ to be equal to the target line number (as a series of ASCII characters, e.g. "100") and call the routine. SEARCH will quickly let you know if it is safe to delete the line.

Perhaps you want to LIST a specific range of lines for possible editing but you can't remember exactly which line to start at. Rather than LISTing the whole program, set 01$ to something you know is contained in the desired range of lines and call the routine. SEARCH takes care of the rest....

Assembly Language Listing

```
                    ********************************
                    *        STRING SEARCH         *
                    ********************************

0E00                        ORG $01DD       in cassette buffer

01DD                SEARCH  EQU *
01DD BDB3ED                 JSR $B3ED       get descriptor addr
01E0 1F02                   TFR D,Y         need A & B, use Y
01E2 E6A4                   LDB ,Y          get length
01E4 10AE22                 LDY 2,Y         point to actual string
01E7 3424                   PSHS B,Y        save both
01E9 5D                     TSTB            length = 0?
01EA 2743                   BEQ SR05        yes, undefined, leave
01EC 9E19                   LDX $19         get start of program

01EE                SR01    EQU *
01EE 6D8C4A                 TST (COUNTR,PCR any occurrences?
01F1 271F                   BEQ SR02        no, skip
01F3 EC42                   LDD 2,U         get line number
```

```
01F5 3440              PSHS U           save pointer
01F7 BDBDCC            JSR $BDCC        go display it
01FA 862D              LDA #'-
01FC BDA282            JSR $A282        followed by a hyphen
01FF E68C39            LDB <COUNTR,PCR  get count
0202 4F                CLRA
0203 BDBDCC            JSR $BDCC        display count
0206 8620              LDA #$20
0208 BDA282            JSR $A282        followed by a space
020B 6F8C2D            CLR <COUNTR,PCR reset new line
020E 3540              PULS U           restore pointer
0210 AEC4              LDX ,U           get addr of next line

0212             SR02  EQU *
0212 1F13              TFR X,U          copy of search addr
0214 EC84              LDD ,X           link to next line
0216 2717              BEQ SR05         end of program, leave
0218 3004              LEAX 4,X         point past link, line #

021A             SR03  EQU *
021A E6E4              LDB ,S           get search length
021C 10AE61            LDY 1,S          get address of string

021F             SR04  EQU *
021F A680              LDA ,X+          get next BASIC byte
0221 27CB              BEQ SR01         end of line, try next
0223 A1A0              CMPA ,Y+         same as search string?
0225 26F3              BNE SR03         keep looking
0227 5A                DECB             checked whole string?
0228 26F5              BNE SR04         loop til done
022A 6C8C0E            INC <COUNTR,PCR bump occurences
022D 20EB              BRA SR03         loop for whole program

022F             SR05  EQU *
022F 860D              LDA #$0D         get a carriage return
0231 BDA282            JSR $A282        display it
0234 3524              PULS B,Y         cleanup stack
0236 4F                CLRA             return a zero to caller
0237 5F                CLRB
0238 7EB4F4            JMP $B4F4

023B 00          COUNTR FCB $00         number of occurrences

023C                   END SEARCH

NO ERRORS FOUND
```

Entry Requirements--SEARCH INIT

1. The address in memory where you want the machine language code to be POKEd must be passed in variable 00.

2. The INIT subroutine must be called at least once before the MAIN subroutine can be called.

Exit Conditions--SEARCH INIT

1. INIT defines USR 0 for use by the MAIN subroutine.

2. All variables are released for re-use.

Entry Requirements--SEARCH MAIN

1. The data to search for must be passed in variable 01$. It may be a string of any length from 1 to 255 characters, containing any combination of ASCII, control, or graphic characters.

Exit Conditions--SEARCH MAIN

1. If the string was found in the body of the program, the line number(s) and the number of occurrences within each line will be displayed on the screen; otherwise, nothing will be displayed.

Variables Used

00, 01, 01$, 02$

Sample Call

```
00010  00 = &H01DD                    * where to put it
       :GOSUB 60000                   * go initialize
       :01$ = "SEARCH"                * target string
       :GOSUB 60100                   * go search
       .
       .                              * more instructions
       .
30000  END
```

Subroutine Listings

```
59999  REM search init
60000  IF PEEK (OO) = &HBD AND PEEK (OO + 93) = &HF4 THEN
       RETURN
60010  O1$ = "BDB3ED1F02E6A410AE2234245D27439E196D8C4A271F
       EC423440BDBDCC862DBDA282E68C394FBDBDCC8620BDA282
       6F8C2D3540AEC41F13EC8427173004E6E410AE61A68027CB
       A1A026F35A26F56C8C0E20EB860DBDA28235244F5F7EB4F400"
60020  FOR O1 = 0 TO 94
      :O2$ = "&H" + MID$ (O1$, 2 * O1 + 1, 2)
      :POKE (OO + O1), VAL (O2$)
      :NEXT
      :DEF USR 0 = OO
      :RETURN

60099  REM search main
60100  O1$ = O1$ + ""
      :O1 = USR 0 (VARPTR (O1$))
      :RETURN
```

Section Five

REFERENCE TABLES

BASIC KEYWORDS

The following tables show the various commands available in each level of BASIC. Note that Extended BASIC may modify the operation of BASIC commands and that Disk BASIC may modify both BASIC and Extended BASIC.

Color BASIC Keywords

*	CLS	LEN	PRINT@
+	CONT	LIST	READ
-	CSAVE	LLIST	REM
/	DATA	MEM	RESTORE
/	DIM	MID$	RETURN
<	END	MOTOR	RIGHT$
=	EOF	NEW	RND
>	EXEC	NOT	RUN
ABS	FOR-NEXT-STEP	ON-GOSUB	SET
AND	GOSUB	ON-GOTO	SGN
ASC	GOTO	OPEN	SIN
AUDIO	IF-THEN-ELSE	OR	SKIPF
CHR$	INKEY$	PEEK	SOUND
CLEAR	INPUT	POINT	STOP
CLOAD	INT	POKE	STR$
CLOADM	JOYSTK	PRINT	USR
CLOSE	LEFT$	PRINT TAB	VAL

Extended Color BASIC Keywords

ATN	EXP	PCLS	SQR
CIRCLE	FIX	PCOPY	STRING$
COLOR	GET	PLAY	TAN
COS	HEX$	PMODE	TIMER
CSAVEM	INSTR	POS	TROFF
DEF FN	LET	PPOINT	TRON
DEF USR	LINE	PRESET	USRn
DEL	LINE INPUT	PRINT USING	VARPTR
DLOAD	LOG	PSET	↑
DLOADM	MID$=	PUT	
DRAW	PAINT	RENUM	
EDIT	PCLEAR	SCREEN	

Disk Color BASIC Keywords

BACKUP	DSKIO$	LOC	SAVE
COPY	FIELD#	LOF	SAVEM
CVN	FILES	LSET	UNLOAD
DIR	FREE	MERGE	VERIFY
DOS*	GET#	MKN$	WRITE
DRIVE	KILL	PUT#	
DSKINI	LOAD	RENAME	
DSKI$	LOADM	RSET	

* The DOS command is available only in Disk BASIC 1.1.

BASIC KEYWORDS BY FUNCTION

The following tables show the various commands available in all BASICs grouped according to function. Note that some keywords may appear in more that one section.

Arithmetic and Logic Operators

+	/	=	OR
-	<	↑	NOT
*	>	AND	

Mathematical and Numeric Operators

ABS	CVN	INT	SIN
ASC	EXP	LOG	SQR
ATN	FIX	RND	TAN
COS	FN	SGN	

String Functions

CHR$	HEX$	LEN	RIGHT$
DSKI$	INSTR	MID$	STR$
DSKO$	LEFT$	MKN$	STRING$

Input/Output Statements and Functions

CLOAD	EOF	LOADM	PRINT@
CLOADM	FIELD#	LOC	PUT#
CLOSE	FREE	LOF	RENAME
COPY	GET#	LSET	RSET
CSAVE	INKEY$	MERGE	RUN
CSAVEM	INPUT	OPEN	SAVE
DIR	JOYSTK	POS	SAVEM
DRIVE	KILL	PRINT	UNLOAD
DSKI$	LINE INPUT	PRINT TAB	WRITE
DSKO$	LOAD	PRINT USING	VERIFY

Graphics Commands and Functions

CIRCLE	LINE	POINT	RESET
CLS	PAINT	PPOINT	SCREEN
COLOR	PCLS	PRESET	SET
DRAW	PCOPY	PSET	
GET	PMODE	PUT	

Memory Commands and Functions

CLEAR	MEM	POKE	VARPTR
EXEC	PCLEAR	TIMER	
FILES	PEEK	USR	

Program Control Statements

CONT	GOSUB	ON-GOSUB	STOP
END	GOTO	ON-GOTO	
FOR-NEXT-STEP	IF-THEN-ELSE	RETURN	

System Commands

BACKUP	DLOAD	LIST	RUN
CLOAD	DLOADM	LLIST	SAVE
CLOADM	DOS*	LOAD	SAVEM
COPY	DRIVE	LOADM	SKIPF
CSAVE	DSKINI	MERGE	TROFF
CSAVEM	EDIT	NEW	TRON
DEL	FREE	RENAME	UNLOAD
DIR	KILL	RENUM	VERIFY

* The DOS command is available only in Disk BASIC 1.1.

Music and Sound Commands

AUDIO	MOTOR	PLAY	SOUND

Miscellaneous Functions and Commands

DATA	DIM	READ	RESTORE
DEF	LET	REM	

BASIC ERROR CODES

This table lists the error messages generated and the internal code used by BASIC in its error routine. When an error is encoutered in interpreting a BASIC statement the error code value is loaded into the B register and program execution is tranfered to the error handling routine at $AC46. The values $00 to $30 are generated by COLOR BASIC, $32 and $34 by EXTENDED COLOR BASIC and $36 to $4A by DISK COLOR BASIC.

/0	$14	DIVISION by ZERO. Division by zero is not possible.
AE	$42	File ALREADY EXISTS. An attempt has been made to RENAME or COPY a file using a filename already used on the disk.
AO	$24	ALREADY OPEN. An attempt was made to OPEN a file (or file buffer) which had already been OPENed.
BR	$36	BAD RECORD number. A record number of 0 or a record number which would make your file greater than 156,672 bytes long was used with a GET# or PUT#.
BS	$10	BAD SUBSCRIPT. The subscript used in an array variable is out of range (greater than that specified by a DIMension command).
CN	$20	CAN'T CONTINUE. A CONT command was used after the end of the program was encountered or after an error, CLEAR, NEW or EDIT.
DD	$12	DUPLICATE DEFINITION. An attempt has been made to redimension an array. If you need to reuse an array variable with a different dimension you will need to use a CLEAR first.
DF	$38	DISK FULL. There is no more room on the disk being written to.
DN	$26	DEVICE NUMBER error. An illegal device number has been used with an OPEN, CLOSE, PRINT, WRITE or INPUT statement.
DS	$30	DIRECT STATEMENT. Usually caused when loading a BASIC program (stored in ASCII) with a missing line number.

ER	$4A	Write or input past END of RECORD. An attempt to INPUT, WRITE or PRINT data to a direct access file without updating the buffer pointer with GET# or PUT#.
FC	$08	Illegal FUNCTION CALL. An illegal parameter has been used with a BASIC command; or a USR call has been attempted before the USR has been defined.
FD	$22	Bad FILE DATA. An attempt was made to read string data from a file while assigning it to a numeric variable.
FM	$2A	FILE MODE error. An attempt has been made to input data from a file opened for output, or output data to a file opened for input.
FN	$3E	Bad FILE NAME. The format used for a disk filename was illegal.
FO	$44	FIELD OVERFLOW error. The length of the string variables used in a FIELD statement exceed the record size specified when opening the file.
FS	$40	FILE STRUCTURE error. The information in the file allocation table and the disk directory do not match.
ID	$16	ILLEGAL DIRECT statement. INPUT or LINEINPUT was used in a direct command line -- they can only be used in a program.
IE	$2E	INPUT past END of file. There is no more data in the file you are reading data from. Avoid this error with EOF.
IO	$28	INPUT/OUTPUT error. The computer cannot read data stored on a peripheral device (tape or disk); usually due to incorrect disk/tape speeds or volume settings or bad media.
LS	$1C	LONG STRING. An attempt has been made to create a string longer than 255 characters.
NE	$34	File does NOT EXIST. The requested disk file doesn't exist or, when using DLOAD or DLOADM, the file doesn't exist in the host computer's memory.

NF	$00	NEXT without FOR. A NEXT has been used without a matching FOR statement. This error will also occur if the variables used with the NEXT and the FOR are not identical.
NO	$2C	NOT OPEN. An attempt has been made to input or output data to a file not yet OPENed.
OB	$3A	OUT of BUFFER. The buffer memory reserved with FILES is not sufficient for an OPEN statement.
OD	$06	OUT of DATA. A DATA entry was not found when a READ statement was interpreted.
OM	$0C	OUT of MEMORY. All the memory accessible to BASIC has been used or allocated.
OS	$1A	OUT of STRING space. There is not enough memory to store the string data or to do a string manipulation.
OV	$0A	OVERFLOW error. The number is outside the range which BASIC can handle.
RG	$04	RETURN without GOSUB. A RETURN has been encountered without a matching GOSUB. Either the GOSUB was not encountered or the stack was reset with an error, CLEAR or PCLEAR.
SE	$46	SET to unfielded string. The variable used in a LSET or RSET statement has not been FIELDed.
SN	$02	SYNTAX error. The BASIC interpreter cannot interpret the statement. Commands may be mis-spelled, required delimiters may be missing or parenthesises may not be matched.
ST	$1E	STRING formula too complex. The level of nesting of string operations was to complex. In our experiments we were unable to generate this error: let us know if you can!
TM	$18	TYPE MISMATCH. An attempt has been made to assign numeric data to a string variable or string data to a numeric variable; or a string expression was used in an argument requiring a numeric expression or a numeric expression was used in an argument requiring a string expression.

UF	$32	UNDEFINED FUNCTION. A function has been called with a FN before it has been defined.
UL	$0E	UNDEFINED LINE. A GOTO, GOSUB, THEN or ELSE is followed by a line number which does not exist in your program.
VF	$48	VERIFY error. If VERIFY is ON and a disk write command encounters an error in reading back the sector just written (verifing) this error will occur.
WP	$3C	WRITE PROTECT error. A command to write data to the disk has been attempted with a "write protect tab" on the disk.

ALPHABETICAL LISTING OF SUBROUTINES

NAME	DESCRIPTION	FIRST LINE #
BAUDRATE	RS232 baud rate initialize	40200
BORDER INIT	Text screen border routines	52100
BORDER MAIN		52200
BREAKDIS	Break key disable	40100
CASSNAME	Input a cassette filename	41000
CHKDRIVE	Check a disk drive	42100
CLOCK INIT	Real time clock routines	51100
ENABLE CLOCK		51200
DISABLE CLOCK		51300
DISPLAY ON/OFF		51400
SET TIME		51500
SET DISPLAY POSITION		51600
DIR	Read/display a disk directory	42200
DISKNAME	Input a disk filename	42300
DPEEK	Double PEEK	40600
DPOKE	Double POKE	40700
EDLIN INIT	Text input routine with editing	52900
EDLIN MAIN		53000
FILEXIST	See if a disk file exists	42400
FLASH INIT	Moving graphics wait routine	50200
FLASH MAIN		50300
GETDATE	Input a date as DD/MM/YY	40900
GRNUMBER INIT	high res graphics number display	40000
GRNUMBER MAIN		40020
HRCHRSET	High res text character set	42700
HRINPUT	High res text input routine	42500
HRPRINT	High res text display routine	42600

NAME	DESCRIPTION	FIRST LINE #
INKEY$	Wait for a single key input	41100
INPUT INIT INPUT MAIN	Text input routine without editing	50000 50100
INVERT INIT INVERT MAIN	Flip text screen values	51700 51800
JOYORKEY	Select joystick or keyboard	40400
JOYSTICK	Read joystick values	40300
KEYINPT	Single key input routine	41200
LINEINPT	Line input routine	41300
MENUDISP	Display a menu	42000
MLSET INIT MLSET MAIN	High speed ML initialization	52300 53400
NEATPRNT	Display text without wordwrap	41810
PACK INIT PACK MAIN	Store a date in two bytes	52700 52800
PCLEAR0	Clear all graphic pages	63900
PRESCON1 PRESCON2	Press any key to continue pause	41500 41600
PRINTON	Check to see printer ready	41700
READY#	Ready an I/O device	41400
RESTORE INIT RESTORE MAIN	Data line restore	52500 52600
SCRAMB INIT SCRAMB MAIN	Scramble the text screen	51900 52000
SCREENPT	Dump text screen to printer	41900
SCROLL INIT SCROLL MAIN	Scroll the screen in any direction	50400 50500

NAME	DESCRIPTION	FIRST LINE #
SEARCH INIT	Find an occurance in a program	60000
SEARCH MAIN		60100
SYSTEM	Get system types/status	40800
TIMER INIT	Interval timer routines	50600
ENABLE TIMER		50700
SET START TIME		50800
GET TIME REMAINING		50900
DISABLE TIMER		51000
TOBASIC	Exit to BASIC	40500

PRINTABLE ASCII CHARACTERS

This table shows all the characters which can be printed on the text screen. The first column shows the decimal value, the second the hexadecimal value and the third the character printed. Characters 8 and 13 (BACKSPACE and CARRIAGE RETURN) do not display a character on the screen, but they are included since they do effect the screen display.

8	$08	BS	55	$37	7	80	$50	P	105	$69	i
13	$0D	CR	56	$38	8	81	$51	Q	106	$6A	j
32	$20	space	57	$39	9	82	$52	R	107	$6B	k
33	$21	!	58	$3A	:	83	$53	S	108	$6C	l
34	$22	"	59	$3B	;	84	$54	T	109	$6D	m
35	$23	#	60	$3C	<	85	$55	U	110	$6E	n
36	$24	$	61	$3D	=	86	$56	V	111	$6F	o
37	$25	%	62	$3E	>	87	$57	W	112	$70	p
38	$26	&	63	$3F	?	88	$58	X	113	$71	q
39	$27	'	64	$40	@	89	$59	Y	114	$72	r
40	$28	(	65	$41	A	90	$5A	Z	115	$73	s
41	$29	)	66	$42	B	91	$5B	[	116	$74	t
42	$2A	*	67	$43	C	92	$5C	\	117	$75	u
43	$2B	+	68	$44	D	93	$5D	]	118	$76	v
44	$2C	,	69	$45	E	94	$5E	[@]	119	$77	w
45	$2D	-	70	$46	F	95	$5F	↑	120	$78	x
46	$2E	.	71	$47	G	96	$60	←	121	$79	y
47	$2F	/	72	$48	H	97	$61	a	122	$7A	z
48	$30	0	73	$49	I	98	$62	b	123	$7B	[[]
49	$31	1	74	$4A	J	99	$63	c	124	$7C	[/]
50	$32	2	75	$4B	K	100	$64	d	125	$7D	[]]
51	$33	3	76	$4C	L	101	$65	e	126	$7E	[↑]
52	$34	4	77	$4D	M	102	$66	f	127	$7F	[←]
53	$35	5	78	$4E	N	103	$67	g			
54	$36	6	79	$4F	O	104	$68	h			

NOTES: 1. The ASCII values 128 to 255 ($80 to $FF) produce various graphics characters.

2. Characters enclosed in square brackets [] and lowercase letters are printed in reverse video.

3. ASCII values 0 to 7 ($00 to $07), 9 to 12 ($09 to $0C) and 14 to 31 ($0E to $1F) have no effect when printed on the screen.

CHARACTERS PRODUCED WHEN VALUES ARE POKED ONTO TEXT SCREEN

This table shows the results of POKEing a value directly onto the text screen -- memory locations 1024 to 1535 ($0400 to $05FF). For example, the statement POKE 1024,2 would display an inverse video "B" in the top left corner of the screen.

0	$00	[@]	32	$20	[space]	64	$40	@	96	$60	space
1	$01	[A]	33	$21	[!]	65	$41	A	97	$61	!
2	$02	[B]	34	$22	["]	66	$42	B	98	$62	"
3	$03	[C]	35	$23	[#]	67	$43	C	99	$63	#
4	$04	[D]	36	$24	[$]	68	$44	D	100	$64	$
5	$05	[E]	37	$25	[%]	69	$45	E	101	$65	%
6	$06	[F]	38	$26	[&]	70	$46	F	102	$66	&
7	$07	[G]	39	$27	[']	71	$47	G	103	$67	'
8	$08	[H]	40	$28	[(]	72	$48	H	104	$68	(
9	$09	[I]	41	$29	[)]	73	$49	I	105	$69	)
10	$0A	[J]	42	$2A	[*]	74	$4A	J	106	$6A	*
11	$0B	[K]	43	$2B	[+]	75	$4B	K	107	$6B	+
12	$0C	[L]	44	$2C	[,]	76	$4C	L	108	$6C	,
13	$0D	[M]	45	$2D	[-]	77	$4D	M	109	$6D	-
14	$0E	[N]	46	$2E	[.]	78	$4E	N	110	$6E	.
15	$0F	[O]	47	$2F	[/]	79	$4F	O	111	$6F	/
16	$10	[P]	48	$30	[0]	80	$50	P	112	$70	0
17	$11	[Q]	49	$31	[1]	81	$51	Q	113	$71	1
18	$12	[R]	50	$32	[2]	82	$52	R	114	$72	2
19	$13	[S]	51	$33	[3]	83	$53	S	115	$73	3
20	$14	[T]	52	$34	[4]	84	$54	T	116	$74	4
21	$15	[U]	53	$35	[5]	85	$55	U	117	$75	5
22	$16	[V]	54	$36	[6]	86	$56	V	118	$76	6
23	$17	[W]	55	$37	[7]	87	$57	W	119	$77	7
24	$18	[X]	56	$38	[8]	88	$58	X	120	$78	8
25	$19	[Y]	57	$39	[9]	89	$59	Y	121	$79	9
26	$1A	[Z]	58	$3A	[:]	90	$5A	Z	122	$7A	:
27	$1B	[[]	59	$3B	[;]	91	$5B	[	123	$7B	;
28	$1C	[\]	60	$3C	[<]	92	$5C	\	124	$7C	<
29	$1D	[]]	61	$3D	[=]	93	$5D	]	125	$7D	=
30	$1E	[↑]	62	$3E	[>]	94	$5E	↑	126	$7E	>
31	$1F	[←]	63	$3F	[?]	95	$5F	←	127	$7F	?

NOTES: 1. All characters in square brackets [] will appear on the screen in inverse video.

2. The values 128 to 255 ($80 to $FF) will produce the same graphics characters as the corresponding PRINT statement.

HEXADECIMAL-DECIMAL CONVERSION CHART

This chart will allow you to convert any 16-bit value from hex to decimal or vice versa.

Hex to Decimal

Obtain the decimal value for each nybble, starting with the most significant nybble. Once all values have been obtained, simply add the values together. For example, to convert the hex number $A0F0, get the value for "A" from the leftmost column, then the "0" from the next column, and so on. The result will be: 40960 + 0 + 240 + 0 = 41200.

Decimal to Hex

Find the largest decimal number in the table which is less than the number being converted. Obtain the hex digit for that value and subtract the value from the number. Treat the remainder as if it were another number to be converted and proceed in exactly the same way. For example, to convert the decimal number 14399, look in the first column to find the largest number less than 14399, which is 12288. This will give you the most significant nybble of what will be a 4-digit hex number, namely "3". Subtracting 12288 leaves a remainder of 2111, which must be converted in the same way. The final hex number will be $383F.

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

The Ultimate Color Computer Reference Guide and Toolkit
Order Form - Subroutines and Utilities

This package includes all the subroutines listed in The Ultimate Color Computer Reference Guide and Toolkit plus the following eight utilities with detailed documentation:

NEATLIST (ML)	Produces neat listings of programs to make them extremely readable by inserting spaces and linefeeds. Output can be directed to screen, printer or a file.
VARXREF (ML)	Lists all occurences of all variables alphabetically by line number.
VARCOUNT (ML)	Gives a listing of the variables used by frequency of usage.
LINEXREF (ML)	Lists all line numbers referenced in GOSUB's, GOTO's, ELSE's, and THEN's
MLAPPEND (ML)	Appends machine language programs to the end of BASIC programs.
BASMUNCH (ML)	Compresses a BASIC program removing all spaces and remarks. It concatenates lines to their maximum possible length, some in excess of 255 bytes long.
BASPROT (ML)	Converts a BASIC program into a pseudo machine language program which you can CLOADM and EXEC. Includes error trapping plus break and reset protection.
DISKMAN (BAS)	Disk file maintenance utility written in structured BASIC to demonstrate the use of subroutines presented in the book.

Name.....

Address.....

City.....Prov/State.....Code.....

Quantity	Description	Total
_____	Subroutines on Tape $12.95 U.S. or $14.95 CDN	$_____
_____	Subroutines on Disk $14.95 U.S. or $17.95 CDN	$_____
	Shipping	+$1.50
	Grand Total	$_____

Payment by: __VISA __Mastercard __Money Order __Certified Cheque

Card #_____

Signature:_____Date(s) on card_____

Note: Additional copies of the book The Ultimate Color Computer Reference Guide and Toolkit may be ordered at $27.95 U.S. or $34.95 CDN plus $3.50 shipping.

CMD Micro
10447-124 St.
Edmonton, Alta. Canada T5N 1R7
Phone (403) 488-7109