MDBASIC

An extension to the Commodore 64 BASIC interpreter

First Publication date: February 14, 1990
Printed in United States of America

If you have any questions about the program MDBASIC or this book, you may write to:

Mark Bowren
mark@bowren.com

The program MDBASIC is dedicated to Kevin Dean Earley.

Table of Contents

PREFACE

This documentation is written for a novice BASIC programmer and is not to be considered an instructional guide for learning programming. Since MDBASIC is an extension to CBM BASIC, only the extensions are documented here. Refer to the Commodore 64 Programmer's Reference Guide for details on CBM BASIC.

There are many different programs on the market that aid the programmer in certain areas of programming such as debugging, graphics, and sound, but there is no program on the market that puts all the features of the Commodore 64 together like MDBASIC. Finally, the full power of the C64 is available to the BASIC programmer.

MDBASIC stands for "Mark Daniel Bowren's All-Purpose Symbolic Instruction Code." It is a program written to coexist with the Commodore 64's BASIC interpreter. It is 100% compatible with standard CBM BASIC. MDBASIC augments some CBM BASIC statements and adds several other additional statements and functions.

MDBASIC consists of 16KB RAM. It is located at memory address $8000 (32768) to $BFFF (49151). It uses 8K of BASIC's RAM ($8000-$9FFF) and 8K RAM that is "underneath" BASIC's ROM ($A000-$BFFF) which leaves 30718 bytes of memory free for BASIC programs. MDBASIC programs will be much shorter than traditional CBM BASIC programs which makes up for the reduction of memory.

MDBASIC provides 60 additional keywords to CBM BASIC, aiding the programmer with I/O, sound, text, sprites, redefined characters, bitmap screens, and programming-debugging. Many keywords combine with other keywords to provide even more capabilities. Some existing CBM BASIC statements have been augmented to provide additional capabilities. Some statements have optional parameters and can be omitted or skipped using commas enabling several variations of functionality. All this reduces the number of keywords to learn instead of having over a 100 like with other BASIC extensions on the market.

MDBASIC includes several other useful features. The aid of the eight function keys that can be individually assigned, hexadecimal ($) binary (%) and octal (@) number representation, memory address range display after loading programs and much more.

I feel once you have used MDBASIC and understand most of the commands you will not want to go back to using standard CBM BASIC.

Sincerely,

Mark Daniel Bowren

INSTALLATION

To run MDBASIC from Commodore 64 BASIC, type the following:

LOAD"MDBASIC.PRG",8,1

SYS64738

MDBASIC will automatically initialize and run just like a cartridge program. The following is a summary of keywords provided or augmented by MDBASIC and categorized by the tasks they perform. Some keywords are used as statements and functions (KEY, ERR, TIME).

		DEBUGGING		
FIND	STOP*	TRACE	VARS	
		UTILITIES		
AUTO	DISK	DUMP	DELETE	FILES
LOAD*	MERGE	NEW*	OLD	RENUM
RUN*	SAVE*	SYS*		
		GRAPHICS		
CIRCLE	COLOR	DRAW	DESIGN	LINE
MAPCOL	MOVE	PAINT	PLOT	SCREEN
SPRCOL	SPRITE	TEXT		
		AUDIO		
ENVELOPE	FILTER	PLAY	PULSE	VOICE
VOL	WAVE			
		MISCELLANEOUS		
CLOSE*	CURSOR	ELSE	ERR	FILL
KEY	ON*	POKE*	RESTORE*	RESUME
RETURN*	SERIAL	SCROLL	SWAP	WAIT*
		FUNCTIONS/VARIABLES		
ERR	ERRL	TRIM	TRIM$	HEX$
INF	INSTR	JOY	KEY	KEY$
MOD	PEN	POT	PTR	ROUND
TIME	TIME$	VALB	VALH	VALO

*An existing CBM BASIC keyword that has been augmented.

Included Software

MDBASIC comes with the following utility programs & games:

CHARACTER EDITOR	Design hires/multicolor characters
SPRITE EDITOR	Design hires/multicolor sprites
POKER	Joker Poker game (written in assembler)
OTHELLO	Board game also known as Reversi
TIC-TAC-TOE	The classic inspired by the movie War Games

FEATURES & ENHANCEMENTS

MDBASIC adds enhanced file loading and saving features described briefly here. See Appendix A for more details.

Device Numbers for loading & saving special data:

- **16** Text screen & colors including background & border
- **17** Character designs (see DESIGN command)
- **18** Bitmaps complete with colors & resolution mode (see SCREEN command)

```
SAVE "MYSCREEN",8,16 :'SAVE CURRENT SCREEN'S TEXT & COLORS TO FILE

LOAD "MYBITMAP",8,18 :'LOAD & SHOW BITMAP FROM FILE
```

You can also save a specific chunk of RAM (binary save):

```
SAVE $0400,$07F8,"MYSCREEN",8  :'SAVE ONLY THE VIDEO MATRIX TO FILE
```

When loading files, the start & end address will be displayed after loading is complete. For example:

```
LOAD"MYPRG",8,1

SEARCHING FOR MYPRG
LOADING 49152-50059
READY.
```

In addition, MDBASIC will load binary files (pictures, machine code, etc.) without corrupting BASIC's memory pointers. A LOAD statement can be in a BASIC program to load binary files anywhere in memory (except $8000-$BFFF which is reserved for MDBASIC) without losing variable data or causing an OUT OF MEMORY error. The programmer must ensure that the target memory locations are available. For example, you would not want to load a file on top of where your program or variables are stored. Instead, manually adjust the BASIC memory pointers with a few POKE statements before loading. The safest approach is to load binaries into upper memory ($C000-$CFFF). Bitmap graphics can be directly loaded into $E000-$FFFF (RAM under Kernel ROM) then shown by using the SCREEN statement.

MDBASIC supports numeric constants of base 2 (binary), 8 (octal) or hexadecimal and new functions have been added to convert the string representation. The NOT expression short-hand is the exclamation point. REM (remark) short-hand is the apostrophe. See examples below:

```
B = %00001111      :'BINARY 15
H1 = $FFFF         :'HEX 65535
H2 = VALH("C000")  :'HEX 49152
H$ = HEX$(65535)   :'DECIMAL 65535 TO HEX STRING FFFF
O = @20            :'OCTAL 16
X = !X             :'SAME AS X = NOT X
```

The LIST command has been enhanced with the ability to freeze the listing process by holding down the shift key. The original behavior only supported slowing down the list process when holding down the control key.

NOMENCLATURE

Direct and Indirect Mode

In direct mode BASIC statements and commands are executed immediately after they are entered on the keyboard. Various uses include performing arithmetic calculations or executing operations on the disk drive. The commands entered are not stored and any variable data is lost when executing a program. Indirect mode is used when entering a program into memory. One or more statements are entered preceded by a line number. The statements are executed using the RUN command. Be sure to save your programs to disk before running newly entered programs.

Keywords, Commands, Statements and Tokens

All these words are essentially referring to the same thing. Keywords are reserved words that cannot be used as a variable. All commands and statements are keywords. When a statement is entered in direct mode it is referred to as a command as it immediately does what you ask. A statement refers to the use of a command in a program. BASIC programs have a binary storage format which uses less space than plain text and executes faster. This is due to the assignment of a single-byte token for each keyword.

Variables, Constants and Literals

A variable is a name given to a known numeric or string value and can be changed at anytime in a running program. Constants are values that do not change. CBM BASIC does not differentiate between named variables and constants. It is the responsibility of the programmer to know which variable names are used as constants and which are variables. Literals are constants without a name. The value is embedded in the program code thus it does not change and therefore must be repeatedly entered when referenced throughout the program.

Expressions and Operations

An expression is one or more variable, constant or literal values of compatible data types combined with operators to yield a single result.

An operation is a process performed on an expression to yield a single result. Operations are performed in a specific order. Parentheses can be used to group expressions to ensure a specific order. The arithmetic order of operations is Parenthesis, Exponents, Multiplication, Division, Addition, Subtraction.

Below are the four types of operations with examples:

1. Arithmetic: +, -, *, /, ^
2. Logical: AND, OR, NOT
3. Relational: =, <, >, <=, >=, <>
4. Functional: EXP(n), SIN(n), STR$(n), VAL(s$), LEFT$(s$,n), etc.

USER REFERENCE GUIDE

Introduction

This section of the manual contains an alphabetical list of commands, statements, functions and variables. Each one is described in detail using the following format:

- ✔ **Purpose** A short summary describing the reason for the instruction
- ✔ **Syntax** Generic variations of how the keywords and parameters are used
- ✔ **Description** A detailed explanation of the instruction and parameters
- ✔ **Example** One or more examples of the instruction

Documentation of Syntax

Each command/statement in this document uses a specific structure to indicate the parameters available and if they are required or optional. Required parameters will cause a SYNTAX ERROR if not present. Optional parameters can be omitted or skipped depending on the statement and may use a default value. The space between commands and parameters is optional and provided for readability. The name of the parameters are also for readability. Actual variable names are limited to only 2 characters.

This document makes use of brackets [] to denote an optional parameter(s). The brackets are not to be entered into the program. When skipping over parameters the comma for each skipped value must be present. When omitting the rest of the optional parameters the statement can be ended normally. For example:

FILTER *[freq] [,resonance] [,type]*

All parameters are optional but at least one must be specified because there are no default values to apply. Statements like the FILES command can omit all parameters since default values are applied for each omitted parameter (*volume$="", device=8*).

DISK *dos$ [,device [,result$]]*

The first parameter is a string and is required. The next two parameters are optional. If the second parameter is supplied then the third parameter can be omitted. The third parameter requires the first two parameters. The following statements are valid syntax based on this definition:

DISK V$: DISK V$,8 : DISK V$,8,S$

The pipe symbol is used to denote alternate syntax. The example below indicates more than one syntax, AUTO ON or AUTO OFF.

AUTO ON | **OFF**

AUTO

PURPOSE:

To automatically generate line numbers with a specified increment while typing in a program.

SYNTAX:

AUTO
AUTO *[increment]*
AUTO ON | OFF

DESCRIPTION:

When entering BASIC program text, every line must start with a line number. Each line number physically follows the next in numeric order but not necessarily by the same increment. AUTO assists the programmer by automatically displaying the next line number that should follow with a predetermined increment. The cursor is positioned after the line number so that BASIC statements can be entered. If the enter key is pressed with no statements on the line then auto-numbering stops. Turn off auto line numbering before editing existing lines of code.

AUTO ON turns on auto-line numbering with the last increment used (default 10).

AUTO OFF (or AUTO 0) turns off auto-line numbering.

increment (0-1023) is an integer value used to calculate the next line number after the enter key is pressed and the cursor is on a line of code. If this parameter is omitted then the last used *increment* (default 10) will be used. A value of zero is the same as AUTO OFF and will not be remembered as the last used *increment*.

The programmer must type in the first line number, or press the return key on the current working line number for the next line to be displayed below the current line. This is not helpful for editing existing lines of code since the line numbers already exist. Turn off auto line numbering before editing existing code. A valid line number is between 0 and 63999.

EXAMPLE:

```
AUTO        :'ENABLES AUTO-LINE NUMBERING USING PREVIOUS INCREMENT (DEFAULT 10)
AUTO 100    :'ENABLES AUTO-LINE NUMBERING WITH INCREMENTS OF 100
AUTO OFF    :'DISABLES AUTO-NUMBERING
AUTO 0      :'DISABLES AUTO-NUMBERING
```

CIRCLE

PURPOSE:

To draw circles or ellipsoids on a bitmap screen.

SYNTAX:

CIRCLE *x, y, xr, yr [,options] [,plotType] [,color]*

DESCRIPTION:

x & *y* are the coordinates for the center of the circle to be drawn. (x=0-319 : y=0-199) hi-res mode; (x=0-159 : y=0-99) multicolor mode. Any values out of range in the mode selected will result in an ILLEGAL COORDINATE ERROR.

xr & *yr* (1-127) define the size of the x radius and y radius. This is the distance from the center point to the outer edge of the circle. Many different ellipsoids can be drawn by varying these values. Values outside this range will cause an ILLEGAL QUANTITY ERROR.

options (0-255, default 15) is an optional set of eight bit flags:

Bit#	Value	Option
0	1	(default) Draw quadrant 1 (top-right)
1	2	(default) Draw quadrant 2 (top-left)
2	4	(default) Draw quadrant 3 (bottom-left)
3	8	(default) Draw quadrant 4 (bottom-right)
4	16	Draw line segment from center to right edge
5	32	Draw line segment from center to top edge
6	64	Draw line segment from center to left edge
7	128	Draw line segment from center to bottom edge

plotType (0-2) is the manner that the dots will be drawn as follows:

```
  0 = Erase dot(s)
  1 = Draw dot(s)
  2 = Flip dot(s) (reverses current condition: on=off, off=on)
  3 = None (useful with the DRAW statement)
```

color (0-15 in hi-resolution mode, 1-3 in multicolor mode), is an optional parameter to select the pixel color. If omitted then the last color selected by any graphics statement will be used. Refer to the MAPCOL statement for details about the available colors for both hires and multicolor bitmap modes. Refer to the COLOR statement for a list of available colors

EXAMPLE:

```
CIRCLE 159,99,50,40,%11111111,1,2  :'DRAW RED CIRCLE WITH ALL OPTIONS
```

CLOSE

PURPOSE:

To close logical files that have been opened.

SYNTAX:

CLOSE *filenum1 [,filenum2 [,filenum3 ...]]*
CLOSE FILES

DESCRIPTION:

CLOSE is the CBM BASIC statement used to close logical files by number. It has been augmented to enable closing multiple files or all open files. If the file number being closed is the current CMD I/O channel then it is reset to use the default I/O devices (keyboard and screen).

filenum (1-255) is the logical file number to close. You can close multiple files by separating them with a comma. No error will occur if the file is not open. The files are closed in the order they appear in the parameter list. CBM BASIC file numbers 128-255 are intended for line printers so each carriage return will be appended with a linefeed character. File number 0 is invalid and ignored without error.

CLOSE FILES will close all logical files and restore default I/O devices. If there were no open files to close then only default devices are restored.

If the file number being closed was opened using device 2 (RS-232 port) then the I/O buffer memory is released and all variable values are lost.

CBM BASIC is limited to a maximum of 10 open files. Any attempt to exceed this value will result in TOO MANY FILES ERROR. You can check the current number of open files using the INF function with parameter value 8. See example below.

EXAMPLE:

```
CLOSE 1       :'CLOSE FILE 1
CLOSE 1,7,9   :'CLOSE FILES 1,7 AND 9
CLOSE FILES   :'CLOSE ALL FILES
IF INF(8) > 0 THEN CLOSE FILES :'CLOSE ALL FILES ONLY IF ANY ARE OPEN
```

COLOR

PURPOSE:

To select the foreground, background and border colors.

SYNTAX:

COLOR *[foreground] [,background] [,border] [,bgcol1] [,bgcol2] [,bgcol3]*

DESCRIPTION:

The COLOR statement is used to select three different color settings for the text screen. Parameters can be omitted to avoid changing the current setting.

foreground (0-15) sets the text color for subsequent PRINT statements.

background (0-15) sets the background color for text and multi-color bitmap screens.

border (0-15) sets the screen border color.

bgcol1, bgcol2 determine the colors used in both multicolor and extended background color text modes. b*gcol3* is only for extended background color mode. For more details on these color modes, refer to the SCREEN statement.

The following is a table of available colors and their associated number:

VALUE	COLOR
0	Black
1	White
2	Read
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow

VALUE	COLOR
8	Orange
9	Brown
10	Light Red
11	Gray 1 (Dark)
12	Gray 2 (Medium)
13	Light Green
14	Light Blue
15	Gray 3 (Light)

EXAMPLE:

```
COLOR 14,6,14   :'SET COMMODORE STANDARD COLOR SCHEME
COLOR ,0        :'SET BACKGROUND COLOR TO BLACK
COLOR 2         :'SELECT RED AS THE COLOR FOR THE NEXT PRINT
```

CURSOR

PURPOSE:

To set the cursor’s visibility and position (column and/or row) on the screen.

SYNTAX:

CURSOR *[column] [,row]*
CURSOR ON | **OFF**
CURSOR CLR

DESCRIPTION:

CURSOR is commonly used in conjunction with the PRINT statement. It allows the cursor to be placed anywhere on a text screen using the video matrix coordinate scheme. It can also be used to control the visibility of the cursor.

column (0-39) represents the column the cursor moves to. If *column* exceeds its range, an ILLEGAL COORDINATE ERROR will result. If omitted then the current column is used.

row (0-24) represents the line the cursor moves to. If *row* exceed its range, an ILLEGAL COORDINATE ERROR will result. If omitted then the current line is used.

CURSOR ON | OFF controls the visibility of the cursor while in program mode.

CURSOR CLR will clear the entire line at the cursor’s current position. You can use this to initialize a line for user input or simply just to clear a specific line by first setting the cursor line then clear it.

EXAMPLE:

```
CURSOR 0,0            :'MOVE CURSOR TO TOP RIGHT CORNER (HOME)
CURSOR 15             :'MOVES CURSOR TO COLUMN 15 ON CURRENT LINE
CURSOR ,10            :'MOVES CURSOR TO LINE 10 OF CURRENT COLUMN
CURSOR ,10:CURSOR CLR :'POSITION CURSOR ON LINE 10 AND CLEARS THE LINE
CURSOR ON             :'MAKE CURSOR VISIBLE AT CURRENT POSITION
CURSOR OFF            :'HIDE THE CURSOR (NO BLINKING)
```

DELETE

PURPOSE:

To remove a line or range of consecutive lines from a program.

SYNTAX:

DELETE
DELETE *line*
DELETE *line1*-*line2*
DELETE *line1*-
DELETE -*line2*

DESCRIPTION:

DELETE aids the programmer when editing a program. Any line or groups of lines may be removed from the program rapidly. Specifying the line numbers to delete is exactly like specifying the lines to display when using the LIST command.

DELETE with no parameters is the same executing the NEW command.

line is a single line number to delete from the program.

line1 is the first line to be deleted in a range. This value may be omitted if the first line to delete is the first line in the program. A dash would follow the number to indicate a range.

line2 is the last line to be deleted in a range. This value may be omitted if the last line to delete is the last line in the program. A dash must always precede this value to indicate a range.

DELETE will not cause an error if the specified line numbers to delete do not exist. If executed in a running program then the program will end.

EXAMPLE:

```
DELETE 40       :'DELETES LINE 40
DELETE          :'DELETES THE ENTIRE PROGRAM
DELETE 150-199  :'DELETES LINES 150 TO 199
DELETE -100     :'DELETES LINES 0 TO 100
```

DESIGN

PURPOSE:

To redefine a character's shape for use on screen pages 1 to 4.

SYNTAX:

DESIGN NEW
DESIGN *screenCode, charset, d0, d1, d2, d3, d4, d5, d6, d7*

DESCRIPTION:

Each character is formed by an 8 x 8 grid of dots, where each dot can be on or off. The data is stored in 8 bytes. The bits that make up each byte will decide what the character looks like. An example of the letter A is shown below:

IMAGE	BINARY	DATA
...**...	00011000	24
..****..	00111100	60
.**..**.	01100110	102
.******.	01111110	126
.**..**.	01100110	102
.**..**.	01100110	102
.**..**.	01100110	102
....	00000000	0

(*=bit on .=bit off)

DESIGN NEW copies all four character sets available in the Commodore 64 ROM. This allows use of the existing character designs while only applying changes to a select few.

Redefined characters are only visible when viewing SCREEN pages 1 to 4, however, new character designs can be applied while viewing any page and are available immediately. If the page has not been displayed since the computer was powered-up then it will contain random text Therefore it is necessary to clear the screen (SCREEN CLR) when switching pages for the first time.

Screen memory and sprite pointers reside together in blocks of 1K RAM based on which page is selected. Switching pages when sprites are visible will result in the loss of the image data. To avoid this copy the image data to the available RAM area that is compatible with the selected page. Use the SPRITE statement to adjust the pointers or just turn off all sprites (SPRITE OFF).

Use the SCREEN statement to select a page for viewing. Pages 1-4 can only support 48 sprite images loaded at one time. All pages share the same color RAM. The table below outlines memory usage for each page.

PAGE	SCREEN RAM	SPRITE POINTERS	AVAILABLE IMAGE RAM	SPRITE DATA INDEXES
0	$0400-$07E7	$07F7-$07FF	$0340-$03FF, $0800-$3FBF	13-15,32-255
1	$C000-$C3E7	$C3F7-$C3FF	$C400-$CFFF	16-63
2	$C400-$C7E7	$C7F7-$C7FF	$C000-$C3FF, $C800-$CFFF	0-15, 32-63
3	$C800-$CBE7	$CBF7-$CBFF	$C000-$C7FF, $CC00-$CFFF	0-31, 48-63
4	$CC00-$CFE7	$CFF7-$CFFF	$C000-$CBBF	0-47

NOTE: Page 3 is used by the bitmap screen. Page 4 is used by the RS-232 I/O buffers. Using SERIAL operations on page 4 will result in the visual display of these buffers. Also, machine language subroutines may need to be relocated.

screenCode (0-255) selects which character to redefine. Note that this is not the ASCII code. The actual character affected is based on the *charset* selected.

charset (0-3) is the character set to modify as listed in the table below:

CHARSET	DESCRIPTION
0	Upper-case and symbols
1	Reverse of set 0
2	Lower-case and symbols
3	Reverse of set 2

d0,d1,d2,d3,d4,d5,d6,d7 are the eight data bytes that make up the shape of each character. By using the percent sign (%), the data can be represented as binary numbers, making obtaining the data simple. If any of the data is omitted, a MISSING OPERAND ERROR will result.

To save the entire redefined character set use secondary device 17, example:

```
SAVE"MYDESIGN",8,17
```

To load a redefined character set use secondary device 17, example:

```
LOAD"MYDESIGN",8,17
```

Saving and loading redefined character sets can be done without being in design mode. Saving without a definition will result in a garbage character set. The memory area for a redefined character set overlaps the second 4K (bottom half) of bitmap memory, thus only one mode can be visible at a time. However this overlap can be used to define a multi-character image using commands for a bitmap (DRAW, LINE, CIRCLE, etc.) then save as a character set. Only dot data would be saved; color is not included and would have to be applied when placing the redefined characters on a text screen.

EXAMPLE:

```
10 DESIGN NEW                   :'COPY ROM CHARS TO RAM FOR INIT DESIGN
20 DESIGN 0,0, 0,0,0,8,8,0,0,0 :'SCAN CODE 0 (CHARACTER @) CHANGES TO A DOT
30 SCREEN 2:SCREEN CLR          :'SWITCH TO PAGE 2 AND CLEAR IT
40 PRINT"@ NOW ON PAGE 2 @"     :'DISPLAY DEMO OF NEW CHAR DESIGN
50 WAIT 240                     :'WAIT 240 JIFFIES (4 SECONDS)
60 SCREEN 0                     :'SWITCH BACK TO STANDARD TEXT SCREEN
```

DISK

PURPOSE:

To send DOS commands to the disk drive(s).

SYNTAX:

DISK *command$ [,device [,result$]]*

DESCRIPTION:

The DISK statement is used to send DOS (Disk Operating System) commands to a selected disk drive, eliminating the need for opening, printing & closing the command channel. All commands offered by the DOS version in the disk drive are available. When executed in immediate mode the status of the command is displayed. In program mode the message is suppressed, however, it can be captured into a string variable.

command$ is a string of characters containing the drive number, function to perform, and the parameters to sent to the drive. An empty string will just report the drive status. The following are the commands offered by the 1541 DOS:

COMMAND	SYNTAX (d=drive number)	DESCRIPTION
NEW	"N*d*:disk name,id"	Full format with ID & label
NEW	"N*d*:disk name"	Soft format (BAM only) with label
COPY	"C*d*:new file=d:original file"	Copy a file
RENAME	"R*d*:new name=old name"	Rename a file
SCRATCH	"S*d*:file name"	Delete a file
INITIALIZE	"I*d*"	Initialize a disk (clear errors)
VALIDATE	"V*d*"	Validate a disk (find problems)

device (8-11, default 8) is the optional device number of the disk drive.

result$ is an optional string variable to store the DOS response message. The response is in the format "Status,Message,Info1,Info2". Below are some examples.

```
00, OK,00,00
00,FILES SCRATCHED,01,00
```

EXAMPLE:

```
DISK"I0"                  :'INITIALIZE DRIVE 0
DISK"N0:JUNK DISK,JD"     :'FORMAT DRIVE 0, NAME="JUNK DISK", ID="JD"
DISK"V"                   :'VALIDATES DISKETTE IN DRIVE 0
DISK"S1:POKER"            :'ERASES PROGRAM "POKER" ON DRIVE 1
DISK"C1:DATA=0:DATA"      :'COPIES "DATA" FROM DRIVE 0 TO DRIVE 1
```

DRAW

PURPOSE:

To draw intricate shapes on a bitmap screen.

SYNTAX:

DRAW *shape$*

DESCRIPTION:

DRAW is used when drawing a picture that has an intricate shape.

The PLOT statement can be used to set the start point, the plot type, and the color of the shape that will be drawn. DRAW always draws from the last plotted point using the last used color & plot type.

shape$ contains the drawing commands separated by commas. Each command has an associated value. Below is the list of available commands:

COMMAND	FUNCTION	VALUE/RANGE
P	Change plot type	0=clear, 1=set, 2=flip, 3=none
C	Change plot color	0-15
U	UP	0-65535
D	DOWN	0-65535
L	LEFT	0-65535
R	RIGHT	0-65535
E	UP & LEFT	0-65535
F	UP & RIGHT	0-65535
G	DOWN & LEFT	0-65535
H	DOWN & RIGHT	0-65535

The values for directional draw commands specify the number of dots (pixels) to draw in the selected direction. If draw reaches the end of the plot area then wrap-around will occur.

To draw without plotting (like lifting the pencil) use plot type 3. This will simply move the current plot coordinates. You will have to set the plot type back to the original value to continue plotting dots.

EXAMPLE:

```
PLOT 10, 10 :'SET CURRENT COORDINATES FOR DRAW
DRAW "C2,R30,P3,R30,P1,R30,D25,L90,U25" :'RED BOX WITH OPEN TOP
```

DUMP

PURPOSE:

To send output to the printer connected on device 4.

SYNTAX:

DUMP LIST *[start]-[stop]*
DUMP SCREEN [*size*]
DUMP VARS
DUMP FILES [*volume$*]
DUMP *[expression]*

DESCRIPTION:

This command is very useful when a hard-copy is needed for a program listing, text or bitmap screen. There are two printer device numbers (4 and 5) but DUMP can only send output to device 4.

DUMP LIST works just like the list routine, except the listing goes on the printer. *start-stop* specify the starting & ending line numbers to print.

DUMP FILES [*volume$*] will print the directory of the disk on the printer.

DUMP SCREEN will print current text screen to the printer. If the currently display screen is a bitmap (SCREEN 5) then the bitmap is printed. The additional parameter *size* (1 = 1x size, 2 = 2x size, default = 1) is available for the bitmap screen only.

DUMP *expression* represents any variable, string or numeric, to be printed on the printer. When not using the other three configurations, DUMP works just like the PRINT command, except the output is only to the printer.

DUMP VARS lists the currently dimensioned variables to the printer.

EXAMPLE:

```
DUMP LIST 100-             :'DUMPS PROGRAM LISTING FROM LINE 100
DUMP SPC(3)+"PRINTER OK."  :'SIMPLE TEST FOR PRINTER
SCREEN 0:DUMP SCREEN       :'DUMPS EVERYTHING ON TEXT SCREEN 0
SCREEN 5:DUMP SCREEN 2     :'PRINT BITMAP SCREEN ENLARGED 2X
```

ELSE

PURPOSE:

To act on a false result of an expression in an IF statement on the same line.

SYNTAX:

IF *expression* **THEN** {*true statements}* **:ELSE** *{false statements}*
IF *expression* **THEN** *line1* **ELSE** *line2*

DESCRIPTION:

If the *expression* result is non-zero (true*)*, all statements between keywords THEN & ELSE will be executed. If the *expression* result is 0 (false), all statements after the ELSE keyword are executed. The ELSE keyword is optional in the IF statement.

The statements THEN and ELSE can be followed by either a line number for branching, or one or more statements to be executed. When THEN is followed by a line number, a colon before the ELSE statement is not needed.

Because IF/THEN/ELSE is all one statement, the ELSE keyword must be on the same line as the IF keyword.

Nesting multiple IF statements is legal but can become confusing. The ELSE fragment is optional for each IF statement. Each ELSE statement corresponds only to the most recent IF statement. In the following example, nothing is printed when A is not equal to B:

```
IF A=B THEN IF B=C THEN PRINT"A=C" : ELSE PRINT"A<>C"
```

By appending another ELSE statement to the end of the above statement, the case when A is not equal to B will be printed:

```
IF A=B THEN IF B=C THEN PRINT"A=C" : ELSE PRINT"A<>C" : ELSE PRINT"A<>B"
```

EXAMPLE:

```
IF A$="N" THEN PRINT"NO":ELSE PRINT"YES" :'YES OR NO WILL BE PRINTED
IF X THEN 100 ELSE 200                   :'GOTO 100 IF X<>0, 200 IF X=0
IF X=1 THEN 200 ELSE X = 0               :'GOTO 200 IF X=1, OTHERWISE SET X=0
```

ENVELOPE

PURPOSE:

To set the duration of the volume envelope phases (attack, decay, sustain, release) for a given voice.

SYNTAX:

ENVELOPE *voice, attack, decay [,sustain, release]*

DESCRIPTION:

When a note is played on a musical instrument, the volume does not suddenly rise to a peak and then cut off to zero. Rather, the volume builds to a peak, levels off to an intermediate value, and then fades away. This creates what is known as a volume envelope.

The SID chip allows the volume envelope of each voice to be controlled so that specific instruments may be imitated or new sounds created. This is done via the attack/decay and sustain/release settings. The cycle is started by using the WAVE statement.

The first phase of the envelope, in which the volume builds to a peak, is known as the attack phase. The second, in which it declines to an intermediate level, is called the decay phase. The third, in which the intermediate level of volume is held, is known as the sustain period. The final interval, in which the sound fades away, is called the release cycle.

voice (1-3) selects which voice to apply the settings.

attack (0-15) is time over which the volume of the tone will rise from 0 to its peak. The 16 durations to choose from set the elapsed time of this cycle as follows:

VALUE	DURATION
0	0.002 seconds
1	0.008 seconds
2	0.016 seconds
3	0.024 seconds
4	0.038 seconds
5	0.056 seconds
6	0.068 seconds
7	0.080 seconds

VALUE	DURATION
8	0.100 seconds
9	0.250 seconds
10	0.500 seconds
11	0.080 seconds
12	1.000 seconds
13	3.000 seconds
14	5.000 seconds
15	8.000 seconds

decay is the time over which the volume of the tone declines from the peak reached in the attack phase to the sustain level. The 16 durations to choose from set the elapsed time of this cycle as follows:

VALUE	DURATION
0	0.006 seconds
1	0.024 seconds
2	0.048 seconds
3	0.972 seconds
4	0.114 seconds
5	0.168 seconds
6	0.204 seconds
7	0.240 seconds

VALUE	DURATION
8	0.300 seconds
9	0.750 seconds
10	1.500 seconds
11	2.400 seconds
12	3.000 seconds
13	9.000 seconds
14	15.000 seconds
15	24.000 seconds

NOTE: *The values in the chart above apply* to both *decay and release* cycles.

sustain (0-15) selects the volume level at which the note is sustained following the decay cycle. The volume of the the voice will remain at the selected sustain level until the release cycle is started using the WAVE statement.

release (0-15) is the duration of time in which the volume fades from the sustain level to near zero volume. The duration of the release cycle corresponds to the values in the decay chart. The cycle is started by using the WAVE statement.

The attack, decay, sustain cycle is started by the WAVE statement by setting the gate operand to 1. The sound being emitted will stay at the sustain volume until the release cycle is started by setting the gate operand to 0.

You may notice the volume of the sound does not quite reach 0 at the end of the release cycle. There are three options to completely turn off the sound to get rid of the residual noise:
1. Set the master volume to zero (VOL statement)
2. Disable the voice's oscillator output (WAVE statement)
3. Route the voice to the no-pass filter (FILTER statement)

EXAMPLE:

```
ENVELOPE 1,1,5           :'VOICE 1 CHANGE ATTACK/DECAY ONLY
ENVELOPE 1,15,15,15,15   :'VOICE 1 MAX VALUES FOR ENVELOPE
ENVELOPE 3,0,0,0,0       :'VOICE 3 MIN VALUES FOR ENVELOPE
```

ERR (statement)

PURPOSE:

To raise an error during program execution. Also used to clear the last error state or turn off error trapping (from ON ERR).

SYNTAX:

ERR *err#*
ERR CLR | OFF

DESCRIPTION:

err# (0-127) is the error number to raise. Error numbers 1-30 are the CBM BASIC errors. Error numbers 31-35 are MDBASIC errors while 0 and 36-127 are user-defined. Any attempt to raise an error outside the valid range will always result in error 14 (Illegal Quantity).

```
 0 USER DEFINED                  18 BAD SUBSCRIPT
 1 TOO MANY FILES                19 REDIM'D ARRAY
 2 FILE OPEN                     20 DIVISION BY ZERO
 3 FILE NOT OPEN                 21 ILLEGAL DIRECT
 4 FILE NOT FOUND                22 TYPE MISMATCH
 5 DEVICE NOT PRESENT            23 STRING TOO LONG
 6 NOT INPUT FILE                24 FILE DATA
 7 NOT OUTPUT FILE               25 FORMULA TOO COMPLEX
 8 MISSING FILENAME              26 CAN'T CONTINUE
 9 ILLEGAL DEVICE NUMBER         27 UNDEF'D FUNCTION
10 NEXT WITHOUT FOR              28 VERIFY
11 SYNTAX                        29 LOAD
12 RETURN WITHOUT GOSUB          30 BREAK
13 OUT OF DATA                   31 MISSING OPERAND
14 ILLEGAL QUANTITY              32 ILLEGAL VOICE NUMBER
15 OVERFLOW                      33 ILLEGAL SPRITE NUMBER
16 OUT OF MEMORY                 34 ILLEGAL COORDINATE
17 UNDEF'D STATEMENT             35 CAN'T RESUME
                                 36-127 USER DEFINED
```

ERR OFF disables error trapping if enabled (See the ON ERR statement).

ERR CLR clears the info from the last error setting ERR=0, ERRL=65535.

EXAMPLE:

```
10 ON ERR RESUME NEXT: PRINT"HERE WE GO"
20 ERR 28 :REM RAISE VERIFY ERROR
30 IF ERRL > -1 THEN PRINT "LAST ERROR#:";ERR;" LINE#:";ERRL
40 ERR CLR : PRINT"ERROR STATE CLEARED"
```

ERR and ERRL (variables)

PURPOSE:

To return information about the last error that was trapped.

SYNTAX:

EN = **ERR**
EL = **ERRL**

DESCRIPTION:

These variables hold information about the last error that occurred. These variables are commonly used in conjunction the ON ERR GOTO statement. When an error occurs during program execution these variables can be interrogated to handle the error based on which one it was and where in the program that it occurred. Refer to the ERR and ON ERR statements details on how to add error handling to a program.

ERR is the error number of the most recent error. If no error occurred then the value is 0 (default).

ERRL is the line number of statement that caused the most recent error. If no error occurred or if the error occurred in immediate mode then the value is 65535 (default) which is not a valid line number.

These variables can be manually cleared (reset to default) using the ERR CLR statement. This will set the default values as ERR = 0 and ERRL = 65535.

EXAMPLE:

```
ON ERR GOTO 1000   :'IF ANY ERRORS OCCUR, GOTO LINE 1000
.
.(main program goes here)
.
1000 PRINT ERR     :'PRINT ERROR NUM
1010 RESUME NEXT   :'IGNORE ALL ERRORS, EXECUTE NEXT STATEMENT
```

FILES

PURPOSE:

To display the directory of the current diskette.

SYNTAX:

FILES
FILES *[volume$] [,device]*

DESCRIPTION:

FILES allows the listing of the directory without loading it into memory. The directory is displayed in the normal CBM DOS directory format with the addition of the total number of files.

FILES, with no parameters, will list the entire directory from drive 0, device 8.

volume$ ("*d*:*volume*") is the search string to filter file names that equal or start with a specific character sequence using the asterisk symbol at the end of the string. The *volume$* parameter can optionally contain the drive number *d*: (0 or 1, default=0) for when two drives are chained together on the same device number. The maximum length of a file name is 16 characters, therefore the maximum length of the *volume$* parameter should never exceed 18 characters.

device (8-11, default 8) is the optional device number of the disk drive.

The listing of the directory can be paused by pressing the shift key, and aborted by pressing the control or break keys. The running file count will be displayed.

EXAMPLE:

```
FILES                   :'ALL FILES ON DRIVE 0, DEVICE 8
FILES,9                 :'ALL FILES ON DRIVE 0, DEVICE 9
FILES"1:*",9            :'ALL FILES ON DRIVE 1, DEVICE 9
FILES"A*"               :'ALL FILES ON DRIVE 0 THAT START WITH A
FILES"0:A*"             :'SAME AS ABOVE
FILES"1:*"              :'ALL FILES ON DRIVE 1, DEVICE 8
FILES"0:*",10           :'ALL FILES ON DRIVE 0, DEVICE 10
```

FILL

PURPOSE:

To fill a section on a text screen with a selected character and/or color.

SYNTAX:

FILL col*1, row1* **TO** *col2, row2 [,poke code] [,color*]

DESCRIPTION:

FILL is used to fill a text screen with a particular character and/or color. A typical application for this statement would be to erase a section of text, or change its color.

col1 & row*1* define the upper left hand corner to start the fill process.

col2 & row*2* define the lower right hand corner to end the fill process.

Text coordinates have the same range as the CURSOR statement. x(0-39) y(0-24). Any value outside this range will result in an ILLEGAL COORDINATE ERROR.

poke code (0-255), AKA: scan code, is the number that represents the character that is to be displayed. The poke code (scan code) is not the same as the ASCII code. This operand can be skipped using a comma in its place, enabling the color to be filled only.

color (0-15) is the color that is to be filled in the defined area of the screen. If in multicolor bitmap mode the color is selected using index values 1,2 or 3. See the PLOT statement for details.

See Appendix C for a list of screen codes. The full list is available in the Commodore 64 Programmers Reference Guide.

EXAMPLE:

```
10 SCREEN 0,0               :'TEXT PAGE 0 WITH NORMAL COLOR SCHEME
20 FILL 0,0 TO 39,0,,2      :'TOP TEXT LINE CHANGES COLOR TO RED
30 FILL 0,24 TO 39,24,32    :'FILL BOTTOM TEXT SCREEN WITH SPACES
40 FILL 10,10 TO 20,20,64   :'FILL CENTER TEXT SCREEN WITH @ SYMBOL
```

FILTER

PURPOSE:

To select filter settings of the Sound Interface Adapter (SID) chip.

SYNTAX:

FILTER *[cutoff] [,resonance] [,type]*
FILTER VOICE *voice [,state]*

DESCRIPTION:

The SID chip supports attenuation (quieter) and amplification (louder) of harmonic frequencies of waveforms from any voice including the A/V port. There is only one filter configuration which can be enabled for any voice.

cutoff (30.0 – 11905.5 Hz) selects the center frequency for making sound quieter the further above and/or below this frequency depending on the type of filter used. SID voices (1-3) can only go as high as 4KHz but voice 4 (input from the AV port, pin 5), can be much higher so the filter supports a higher range.

resonance (0-15) allows peaking the volume of harmonic frequencies nearest the *cutoff* frequency which creates a sharper filtering effect.

type (0-4) selects the type of filter used with the center frequency:
0=no pass -mute entire oscillator output
1=low pass -suppress frequency components above the *cutoff* frequency
2=band pass -suppress frequency components above & below the *cutoff* frequency
3=high pass -suppress frequency components below the *cutoff* frequency
4=band stop -suppress frequency components nearest the *cutoff* frequency

High & low pass filters attenuate the volume of the frequency components furthest away from the *cutoff* frequency by 12dB per octave.

The band pass filter attenuates the volume of the frequency components furthest away from the *cutoff* frequency by 6dB per octave.

The band stop (notch reject) filter attenuates the volume of the frequency components nearest the *cutoff* frequency by 12dB per octave.

FILTER VOICE *voice (1-4), [state: 0=off, 1=on (default)]* controls the filtering of a voice. Voice 4 is the external input on pin 5 of the A/V port.

EXAMPLE:

```
FILTER 1200.5,15,1  :'LOW PASS, CUTOFF=1200.5HZ, RESONANCE 15
FILTER 500          :'JUST CHANGE THE CUTOFF/CENTER FREQUENCY
FILTER ,0           :'JUST TURN OFF RESONANCE
FILTER VOICE 4      :'APPLY FILTER ON EXTERNAL INPUT OF A/V PORT PIN 5
```

FIND

PURPOSE:

To search a BASIC program for keywords or a sequence of characters.

SYNTAX:

FIND"*string*
FIND*code*

DESCRIPTION:

FIND is used for searching program text (in memory) for a specific keyword or string sequence. Large programs can be difficult to manually find things like what strings have been used, statements that have been used, or a miscellaneous string of characters. FIND is a fast, useful debugging tool for programmers.

"string is used when searching for ASCII text (not keywords). The ending quotation mark is left off unless it is part of the string being sought.

code is used when searching for statement keywords. The exact syntax (including any spaces & quotations) will be included in the search.

When FIND makes a match, the line is displayed and the search process continues until the end of the program is reached. If the BREAK key is pressed during a search, the process stops immediately and displays the BREAK IN {line} message indicating where the search was when it was interrupted.

NOTE: All statements following the FIND statement will be included in the search. Every character or statement will be searched, including REM, colons, quotations, spaces etc.

EXAMPLE:

```
Find all occurrences of string assignment (no spaces)
FINDA$=

Find all occurrences of keyword GOSUB with space then digits 200
FINDGOSUB 200

Find all occurrences of exact string
FIND"METEOR
```

TRIM()

PURPOSE:

To return the value of a signed 32-bit single-precision floating-point number with its fractional portion removed (truncated).

SYNTAX:

I = **TRIM(***n***)**

DESCRIPTION:

TRIM returns the whole number (integer) portion of a signed 32-bit floating-point decimal number. This differs from the INT function which returns the lowest (floor) integer value. See the examples below which demonstrates this concept.

n is the 32-bit, single-precision, signed floating-point decimal number to remove (truncate) the fractional portion. If the data type of this parameter is an integer then the result is a float with the same value (unchanged).

To obtain the fractional portion of a number simply subtract the trimmed portion from the original portion as demonstrated in the example below. The ROUND function can be used to correct inaccuracies due to the use of single-precision floating-point numbers.

EXAMPLE:

```
'TRUNCATE VS FLOOR ON NEGATIVE VALUE
N=-4.25 : PRINT TRIM(N), INT(N)
-4        -5

'GET FRACTIONAL PORTION
N=11.0625 : PRINT A-TRIM(N)
.0625

'GET FRACTIONAL PORTION WHEN PRECISION LOSS OCCURS
N=104.0125 : F=N-TRIM(N) : PRINT F, ROUND(F,4)
.0124999881         .0125

10 'PERFORM X MOD Y CALCULATION
20 X=25 : Y=12
30 M = X-TRIM(X/Y)*Y
40 PRINT X;"MOD";Y;"=";M
```

TRIM$()

PURPOSE:

To return the value of a string with all leading and trailing spaces removed.

SYNTAX:

F$ = **TRIM$(***S$***)**

DESCRIPTION:

TRIM$ returns a new string copied from the source string S$ with all leading and trailing spaces removed. If the source string contains only spaces then the result is an empty string.

This functionality is commonly referred to as trimming the string, thus many other programming languages use the keyword TRIM. It is especially useful when dealing with user inputted strings where the value may be sloppy.

The CBM BASIC statements to perform this function is a bit ugly and would not perform as well because it involves searching both sides of the string for the first non-space character and then returning the MID$ result.

Use this function sparingly since a new string is created on each use. Heavy use can trigger a garbage collection. This also happens when assigning one string to the value of another. When heavy use is needed, like inside a loop reading a file, be sure to trigger a garbage collection manually by using SYS46374 or the function FRE(0) so that it does not build up a huge pile which can take a long time to process.

EXAMPLE:

```
0 '*TEST TRIM OF LEAD & TRAIL SPACES*
10 S$=" THIS IS A TRIM$ TEST  "
15 PRINT"SOURCE STRING:"
20 PRINT CHR$(34);S$;CHR$(34)
25 PRINT"LENGTH:";LEN(S$)
30 F$=TRIM$(S$)
35 PRINT:PRINT"RESULT STRING:"
40 PRINT CHR$(34);F$;CHR$(34)
45 PRINT"LENGTH:";LEN(F$)
```

HEX$()

PURPOSE:

To convert a 32-bit signed integer into its hexadecimal string equivalent.

SYNTAX:

H$ = **HEX$(***n***)**

DESCRIPTION:

HEX$ is a function that returns the hexadecimal equivalent of the numeric expression within parenthesis. The Hexadecimal numbering system has a base of 16, rather than base 10 of the Decimal numbering system.

n is the 32-bit signed integer to convert to a hexadecimal string of characters. If *n* is a floating point number then the fractional portion is ignored. The maximum positive value is 2147483647 which equals $7FFFFFFF. The most significant bit (31) is considered the sign flag, therefore negative values range from -1 ($FFFFFFFF) to -2147483646 ($80000001).

EXAMPLE:

```
'CONVERT DECIMAL CONSTANT TO HEX STRING, PRINT RESULT
PRINT HEX$(49152)
C000

'CONVERT DECIMAL VARIABLE VALUE INTO HEX STRING VARIABLE
V$ = HEX$(V)

'CONVERT BINARY CONSTANT TO HEX STRING, PRINT RESULT
PRINT HEX$(%10010)
12

'CONVERT OCTAL CONSTANT TO HEX STRING, PRINT RESULT
PRINT HEX$(@777)
1FF

'CONVERT A BINARY STRING TO DECIMAL THEN HEX STRING THEN PRINT RESULT
B$="10010" : D=VALB(B$) : H$=HEX$(D) : PRINT B$,D,H$
10010      18      12

'CONVERT AN OCTAL STRING TO HEX STRING
O$="172" : H$=HEX$(VALO(O$))
```

INF()

PURPOSE:

To return various system information.

SYNTAX:

I = **INF(***n***)**

DESCRIPTION:

n (0-67) selects the piece of system information. Most information can be read directly from memory registers using PEEK statements. This function helps reduce the use of PEEK statements so that programmers do not have to concern themselves with which memory locations hold the data and how to parse it. For example, INF 37 to 51 return color values ensuring the range 0-15. All sprite X-coordinates return the 9-bit value. INF 21 to 36 return a 16-bit value. Some values are only applicable when a program is running.

VALUE	INFORMATION
0	Cursor physical column number (0-39)
1	Cursor logical column number (0-79) same as POS function
2	Cursor physical line number (0-24)
3	Cursor blink disabled: 0=no, 1=yes
4	Maximum logical column number for current line (39 or 79)
5	ASCII of character under cursor (while blinking)
6	ASCII of last char printed to screen
7	Current command channel file number (0=none, set by CMD statement)
8	Number of open files (max 10)
9	Number of chars in keyboard buffer (max 10)
10	Address of first column of the current cursor line
11	Shift/Ctrl/Logo key bit flags: 1=Shift, 2=Logo, 4=Ctrl
12	Character color (0-15) under cursor while blinking
13	Current foreground color (0-15) for text
14	PAL or NTSC video system, 0=NTSC, 1=PAL
15	Kernal version identification number (0,3,67,100 or 170) Kernal Version System 170 v1 C64 0 v2 C64 3 v3 C64 67 SX C64SX 100 4064 PET 64/Educator

VALUE	INFORMATION
16	Current string index (0-255) of the note playing with PLAY statement
17	Current octave (0-8) of the note playing with PLAY statement
18	Last plotted X coordinate (0-319 hires, 0-159 multicolor)
19	Last plotted Y coordinate (0-199)
20	Last plotted coordinate pixel state (0=pixel off, 1=pixel on)
21	Current VIC-II 16KB memory bank (0 to 3)
22	Start address of the last loaded file
23	End address of the last loaded file
24	Address of the start of BASIC program text
25	Address of the start of the BASIC variable Storage area
26	Address of the start of the BASIC array storage area
27	Address of the end of the BASIC array storage area/start of free RAM
28	Address of the bottom of the string text storage area
29	Address of the last temporary string used by the string builder
30	Highest address used by BASIC
31	Current BASIC line number (0-63999 while running, else immediate mode)
32	Previous BASIC line number after STOP or BREAK in program
33	Address of the current BASIC statement
34	Last read DATA line number (0-63999)
35	Address of the current DATA item inside BASIC program text
36	Address of the source of GET, READ, or INPUT information
37	Border color (0-15) for screen
38	Background color (0-15) for screen with foreground bit pattern 0 All multicolor modes with bit pattern 00
39	Color (0-15) for multicolor text with bit pattern 01 and Extended background color for characters having screen codes 64-127
40	Color (0-15) for multicolor text with bit pattern 10 and Extended background color for characters having screen codes 128-191
41	Color (0-15) for multicolor text with bit pattern 11 and Extended background Color for characters having screen codes 192-255
42	Sprites (0-7) color (0-15) for multicolor bit pattern 01
43	Sprites (0-7) color (0-15) for multicolor bit pattern 11
44	Sprite 0 color for hires bit pattern 1 and multicolor bit pattern 10
45	Sprite 1 color for hires bit pattern 1 and multicolor bit pattern 10
46	Sprite 2 color for hires bit pattern 1 and multicolor bit pattern 10
47	Sprite 3 color for hires bit pattern 1 and multicolor bit pattern 10
48	Sprite 4 color for hires bit pattern 1 and multicolor bit pattern 10
49	Sprite 5 color for hires bit pattern 1 and multicolor bit pattern 10
50	Sprite 6 color for hires bit pattern 1 and multicolor bit pattern 10
51	Sprite 7 color for hires bit pattern 1 and multicolor bit pattern 10
52	Sprite 0 X coordinate (0-511)
53	Sprite 0 Y coordinate (0-255)
54	Sprite 1 X coordinate (0-511)

VALUE	INFORMATION
55	Sprite 1 Y coordinate (0-255)
56	Sprite 2 X coordinate (0-511)
57	Sprite 2 Y coordinate (0-255)
58	Sprite 3 X coordinate (0-511)
59	Sprite 3 Y coordinate (0-255)
60	Sprite 4 X coordinate (0-511)
61	Sprite 4 Y coordinate (0-255)
62	Sprite 5 X coordinate (0-511)
63	Sprite 5 Y coordinate (0-255)
64	Sprite 6 X coordinate (0-511)
65	Sprite 6 Y coordinate (0-255)
66	Sprite 7 X coordinate (0-511)
67	Sprite 7 Y coordinate (0-255)

EXAMPLE:

```
0 'GET COORDINATE OF SPRITE 0
10 SPRITE 0,1,15,0,13
20 POKE832 TO 832+63,255
30 MOVE 0,260,100
40 PRINT"SPRITE 0 IS AT (";INF(51);",";INF(52);")"
50 PRINT"SPRITE 0 HAS COLOR";INF(43)
```

For more examples refer to Appendix E.

INSTR()

PURPOSE:

To find the index of the first occurrence of a string that is inside another string.

SYNTAX:

I = **INSTR(***[index,] source$, find$***)**

DESCRIPTION:

INSTR is a function that returns the index of the position of a string within another string. The index of a string starts at 1. If no match is found the value of -1 is returned.

index (1-255) is an optional numeric expression of where in the string to start the search. If not supplied the start index is 1.

source$ is the string to search.

find$ is the string being sought.

NOTE: Using INSTR in immediate mode with both *source$* and *find$* string parameters as literal string values will produce incorrect results when the find string is found at the end of the source string. This is due to the fact that both strings are temporary strings (copied from the command line) which causes the find string to overlap the end of the source string during evaluation. This can be avoided by using a variable for at least one of the string parameters. This will not happen in program mode since the literal string values in the program text are used during evaluation.

EXAMPLE:

```
10 S$="MDBASIC IS COOL":F$="IS"
20 I%=INSTR(S$, F$)
30 IF I%=0 THEN PRINT"NOT FOUND!":ELSE PRINT MID$(S$, I%, LEN(F$))
```

JOY()

PURPOSE:

To return the status of a joystick in either port.

SYNTAX:

J = **JOY(***port***)**

DESCRIPTION:

port (1-2) selects the input port that the joystick is plugged in.

J is the variable that will be assigned to the current position of the stick. The number returned will be from 0-10 with no button pressed, and 128-138 if the button is pressed. To remove the button state append the logical operation AND 15. The number returned corresponds to a direction in the table below.

NUMBER	DIRECTION
0	no direction
1	up
2	down
3	--n/a--
4	left
5	up & left
6	down & left
7	--n/a--
8	right
9	up & right
10	down & right

Refer to the example program at the end of Appendix E.

EXAMPLE:

```
REM DISPLAY MSG IF JOYSTICK #1 IS PUSHED DOWN (IGNORE BUTTON)
IF JOY(1) AND 15 = 2 THEN PRINT"DOWN"

REM DISPLAY MSG IF JOYSTICK #2'S BUTTON IS PRESSED
IF JOY(2)>10 THEN PRINT"FIRE"

REM ACT ON ALL MOVEMENT-NOTE 3 & 7 ARE N/A SO LINE 0 WILL SUFFICE
J% = JOY(1) AND 15 :REM ALL EXCEPT FIRE BUTTON
IF J% THEN ON J% GOSUB 100,110,0,130,140,150,0,170,180,190
```

KEY (statement)

PURPOSE:

To assign one of the 8 function keys, display the current function key assignments, put characters in the keyboard buffer, clear the buffer and wait for any single key input.

SYNTAX:

KEY *keynum, assign$*
KEY *string$*
KEY LIST | OFF | CLR
KEY GET *variable*
KEY WAIT *[variable]*

DESCRIPTION:

The KEY statement has various forms and uses in both direct and immediate mode. Function key assignments work only in immediate mode.

keynum (1-8) selects which function key to assign text.

*assign$* is the string of characters (maximum 31) that will be assigned to the function key. Adding +CHR$(13) at the end of this string will act as if the user pressed the return key and cause the command to be executed.

string$ (0 to 255 characters) is a string of characters to put into the keyboard buffer as if the user entered them.

KEY LIST displays the current key assignments in immediate mode.

KEY OFF turns off key trapping. See the ON KEY statement for more info.

KEY CLR clears the keyboard buffer.

KEY GET *variable* fetches a key from the keyboard buffer. If the buffer is empty then the *variable* will be zero or an empty string depending on the type.

KEY WAIT waits for a key to appear in the keyboard buffer with an optional *variable* for storing the ASCII value. When no *variable* is provided the key will remain in the buffer and the program will continue to the next statement.

EXAMPLE:

```
KEY 1,CHR$(147)+"LIST"+CHR$(13)  :'ASSIGN FUNCTION KEY 1 TO LIST PROGRAM
KEY "N":INPUT A$                 :'INPUT WITH DEFAULT USER RESPONSE
KEY WAIT K: PRINT K, CHR$(K)     :'WAIT FOR A KEY, PRINT ASCII AND CHR
```

KEY and **KEY$** (variables)

PURPOSE:

To return the ASCII value of the last key pressed when key trapping is enabled.

SYNTAX:

K = **KEY**
*K$* = **KEY$**
KEY = *ascii*

DESCRIPTION:

These variables are used in conjunction with the ON KEY GOSUB statement to return the ASCII value of the key that was pressed. The value will be retained until the next keystroke is trapped. During subroutine execution key trapping is paused until a RETURN statement is executed. It is turned off entirely any time the KEY OFF statement is executed.

The KEY variable can be manually changed to any ASCII value (0 to 255). This is useful when filtering keys during evaluation of the ASCII. Changing the value of KEY immediately changes KEY$ which cannot be changed directly. See example below.

EXAMPLE:

```
0 '***ON KEY GOSUB EXAMPLE***
1 'GET KEY STROKES WHILE MAIN LOOP RUNS
2 'PRG ENDS WHEN ENTER KEY PRESSED
10 CURSOR ON
20 PRINT">";
30 ON KEY GOSUB 100
40 '**YOUR MAIN PROGRAM LOOP STARTS HERE**
50 WAIT1:WAIT1:WAIT1:WAIT1:WAIT1 :REM SIMULATE RUNNING PRG
60 GOTO 40
70 '**NO KEY TRAP SECTION OF PRG GOES HERE**
80 CURSOR OFF
85 PRINT:PRINT"DONE."
90 END
100 CURSOR OFF
105 KEY = (KEY AND 127) : IF KEY = 13 THEN KEY OFF : RETURN 70
110 IF KEY<32 THEN KEY=0
115 PRINT KEY$; : CURSOR ON : RETURN
```

LINE

PURPOSE:

To draw a line between two selected points on a bitmap screen.

SYNTAX:

LINE *x1, y1* **TO** *x2, y2, [plotType], [color]*

DESCRIPTION:

x1 & *y1* define the start point to begin drawing the line.

x2 & *y2* define the end point of where the line ends.

The values for both sets of points are limited and based on the color mode. In hires mode *x1* and *x2* (0-319), *y1* and *y2* (0-199). In multicolor mode *x1* and *x2* (0-159), *y1* and *y2* (0-99). Specifying a point outside the range of the current color mode will result in an ILLEGAL COORDINATE ERROR.

plotType (0-3) determines how the line will be plotted. 0=dots off, 1=dots on, 2=flip pixel (on=off, off=on), 3=none (set plot location).

color (0-15 in hi-resolution mode, 1-3 in multicolor mode), is an optional parameter to select the paint color. If omitted then the last color selected by any graphics statement will be used. Refer to the MAPCOL statement for details about the available colors for both hires and multicolor bitmap modes. Refer to the COLOR statement for a list of available colors.

EXAMPLE:

```
LINE 0,0 TO 319,199      :'DRAWS DIAGONAL LINE ACROSS SCREEN
LINE 160,100 TO 0,199,2  :'FLIPS LINE OF DOTS FROM CENTER TO LOWER LEFT
```

LINE INPUT

PURPOSE:

To input a string of characters from the keyboard that is terminated by a carriage return.

SYNTAX:

LINE INPUT [*"prompt",*] *A$ [,B$, C$,…,Z$]*

DESCRIPTION:

LINE INPUT is much like the INPUT statement, except LINE INPUT accepts input of any character (except return key). This is useful for inputting strings that may contain commas or semicolons, or when no prompt is needed.

"prompt" (optional) is a string of characters that will be displayed as a message to the user. This string cannot be a variable.

A$* is the variable that the string of characters that will be displayed as a message to the user. It is legal to supply multiple strings (*B$, C$,…,Z$) each separated by a comma to capture multiple lines in one statement.

The return key is the only delimiter for the end of the input string. The input from the keyboard it is limited to 80 characters.

EXAMPLE:

```
LINE INPUT A$           :'NO PROMPT, ONE STRING
LINE INPUT A$, B$       :'NO PROMPT, THEN ENTER 2 STRINGS
LINE INPUT"->", A$, B$  :'PROMPT ONCE, THEN INPUT 2 STRINGS
```

LINE INPUT#

PURPOSE:

To input a line of characters from an open file that is terminated by a carriage return.

SYNTAX:

LINE INPUT# *fileNumber*, *A$ [,B$, C$,…,Z$]*

DESCRIPTION:

LINE INPUT# is much like the INPUT# statement, except LINE INPUT# is only terminated by the carriage return character (ASCII 13). This is useful for reading text files that may contain commas or semicolons.

fileNumber is the already opened logical file number to read characters from instead of the keyboard.

A$* is the variable that the string of characters that will be displayed as a message to the user. It is legal to supply multiple strings (*B$, C$,…,Z$) each separated by a comma to capture multiple lines in one statement.

The text file is read starting from the current position to the next carriage return character or if the length of the string reaches 255 characters.

EXAMPLE:

```
10 '***READ FROM FILE***
15 DIM A$,L,C
20 OPEN 1,8,0, "MYFILE.SEQ"
30 LINE INPUT#1, A$
40 PRINT A$
45 L=L+1:C=C+LEN(A$)
50 IF ST AND 64 GOTO 70
60 GOTO 30
70 CLOSE 1
80 PRINT L;"LINES,";C;"CHARACTERS."
90 END
```

MAPCOL

PURPOSE:

To set the default colors to be used when plotting dots on a bitmap screen.

SYNTAX:

MAPCOL [*c1], [,c2] [,c3] [,c4]*

DESCRIPTION:

MAPCOL is used with bitmap graphics for selecting the colors to be used when plotting dots. The next plotted dot by any graphics statements will use the newly applied setting. Refer to the COLOR statement for a list of the available colors.

c1 (0-15) changes the default color for plotting dots. In multicolor mode this is the dot color for bit pattern 01.

c2 (0-15) changes the default background color of the 8 x 8 square that contains the hires dot. In multicolor mode this is the dot color for bit pattern 10.

c3 (0-15) is only used in multicolor mode to set the color for bit pattern 11.

c4 (0-15) is only used in multicolor mode to set the color for bit pattern 00. This is the same color used by the text background.

In hires mode, every 8 x 8 square can only have one color of plotted dots. When mixing colors on a hires screen, each set of 64 dots must be the same color. This a limitation of the VIC-II chip.

In multicolor mode the horizontal resolution is cut in half to support 3 different pixel colors in the same 8 x 8 square. When in this mode, all graphics statements that select a color will use values 1, 2 or 3 corresponding to *c1*, *c2* and *c3* respectively using the following bit patterns:

COLOR	PATTERN	DESCRIPTION
c1	01	Upper nybble of scan code in Video Matrix ($C800-$CBE8)
c2	10	Lower nybble of scan code in Video Matrix ($C800-$CBE8)
c3	11	Lower nybble of Color RAM in Video Matrix ($D800-$DBE8)
c4	00	Background Color Register 0 BGCOL0 ($D021)

EXAMPLE:

```
MAPCOL 1        :'CHANGES C1 ONLY
MAPCOL 1,2      :'CHANGES C1 AND C2 ONLY
```

MERGE

PURPOSE:

To merge a program from disk or tape to the end of a program in memory.

SYNTAX:

MERGE *filename$ [,deviceNum]*

DESCRIPTION:

MERGE allows the combining of programs to constitute one program in memory. The process merges the two programs into RAM memory only. If a new file is to be created, the final product must be saved to a new file.

Before two programs are merged, the first program must be in RAM memory, which will be the beginning of the program. The second program must be on tape or disk.

Be sure to renumber both programs separately so that when they merge together the line numbers do not overlap. The merging program's start line number should be higher than the last line number of the program in memory. Refer to the RENUM command for more information.

filename$ is the file name of the program that is to be connected to the end of the program in RAM memory. For tape devices the string can be empty.

deviceNum (0-31, default 1) is the device number from where the merging program will load. Device 1 is for tape, devices 8 to 12 are for disks.

When MERGE is executed with no parameters the default behavior of LOAD is used which is to MERGE the first file found on the tape device.

NOTE: No secondary address is specified because MERGE is only for BASIC programs.

EXAMPLE:

```
MERGE"SUBROUTINE",8
```

MOD()

PURPOSE:

To return the remainder of a division operation.

SYNTAX:

R = **MOD(***dividend, divisor***)**

DESCRIPTION:

MOD (modulo) is a function that returns the remainder of a division operation. This implementation uses truncated division to determine the quotient. The remainder has the same sign as the dividend.

MOD is typically used with positive integers however fractional values can be used. The range of values for an integer modulo operation of *n* is 0 to (*n* − 1). Any integer n MOD 1 is always equal to 0. n^x mod *n* = 0 for all positive integer values of *x*.

dividend (numerator) is the number to be divided.

divisor (modulus) is the number used to divide the numerator.

The following is the equivalent BASIC expression:

```
    R = X-TRIM(X/Y)*Y
```

It is common for BASIC to define functions for a specific modulus calculation. For example, to calculate X MOD 12 the following function definition can be used:

```
    DEF FN M(X) = X-TRIM(X/12)*12
    PRINT FN M(25)
```

The MOD function simplifies the program text and is easier to understand when reading program text.

EXAMPLE:

```
10 R = MOD(25, 12)  :'25/12 = QUOTIENT 2 REMAINDER 1, OR 2 AND 1/12
20 PRINT R
```

MOVE

PURPOSE:

Moves a sprite to any location or from one location to another at a given speed.

SYNTAX:

MOVE *spriteNum, [x], [y]*
MOVE *spriteNum, x1, y1* **TO** *x2, y2, [speed]*
MOVE *spriteNum* **TO** *x, y, [speed]*

DESCRIPTION:

Any of the eight sprites can be displayed anywhere on the screen or move from one point to another at a selected speed.

spriteNum (0-7) defines which sprite to move.

x (0-511) & y (0-255) select the coordinates for the sprites upper left corner. The coordinates for sprites do not align to bitmap coordinates. A sprite can be placed off the visible screen. To place a sprite at the top-left corner of the visible screen use coordinates x=24 and y=50.

A sprite can be placed at a specific coordinate or move to a specified coordinate from its current location at a specified *speed*.

x1, y1 & *x2, y2* allows a sprite to move from point x1, y1 to point x2, y2. These values have the same range as x and y stated above. Any coordinate value that exceeds the legal range will cause an ILLEGAL COORDINATE ERROR.

speed (0-255) controls the amount of delay in movement between coordinates when the sprite moves from between points with 0 being the fastest. The next BASIC statement will not execute until the movement is complete.

EXAMPLE:

```
MOVE 0,24                 :'ONLY SET X COORDINATE OF SPRITE 0 TO 24
MOVE 0,,100               :'ONLY SET Y COORDINATE OF SPRITE 0 TO 100
MOVE 1,180,120            :'PUT SPRITE 1 AT CENTER OF SCREEN
MOVE 0,24,50 TO 320,229   :'MOVES FROM TOP-LEFT TO BOTTOM RIGHT FAST
MOVE 7 TO 180,100, 50     :'MOVES SPRITE 7 FROM CURRENT TO CENTER SCREEN
```

NEW

PURPOSE:

To reset BASIC memory and discard the BASIC program or to reset the computer.

SYNTAX:

NEW
NEW SYS

DESCRIPTION:

NEW is the CBM BASIC command for removing the current BASIC program from memory. MDBASIC has augmented it to provide a system reset. If NEW is followed by the SYS keyword then the system performs a warm-start. This is equivalent to using SYS64738 or pressing the reset button on your computer. It is provided convenience.

If a program has been NEW-ed or warm-start performed you can restore the previous BASIC program by using the OLD command.

EXAMPLE:

```
NEW        :'THIS WILL REMOVE THE BASIC PROGRAM
NEWSYS     :'THIS WILL PERFORM A WARM RESTART OF THE COMPUTER
```

OLD

PURPOSE:

To restore a BASIC program that has been erased by the NEW command or system reset.

SYNTAX:

OLD

DESCRIPTION:

OLD restores the BASIC program text that has been lost due to using one of the following:

- NEW
- NEW SYS
- SYS 64738
- Electronic reset button

The program text cannot be restored if any of the following has occurred:

- A variable was created
- Another program was entered or loaded
- A POKE statement corrupted memory

EXAMPLE:

```
NEW      :'PROGRAM IS GONE
LIST     :'VERIFY PROGRAM IS GONE
OLD      :'PROGRAM IS RESTORED
LIST     :'VERIFY PROGRAM IS RESTORED
NEWSYS   :'RESET COMPUTER
OLD      :'PROGRAM IS RESTORED
LIST     :'VERIFY PROGRAM IS RESTORED
```

ON ERR

PURPOSE:

To enable and define the global error handling scheme to ignore or use a custom error handling subroutine.

SYNTAX:

ON ERR GOTO *lineNum*
ON ERR RESUME NEXT
[ON] ERR OFF

DESCRIPTION:

ON ERR enables redirection of program execution to a specified line number when an error occurs. This is useful when an error like DEVICE NOT PRESENT occurs, which can be trapped to avoid program crashes.

lineNum is the line number that the program will GOSUB to when an error occurs. An invalid *lineNum* will result in an UNDEF'D STATEMENT ERROR.

ON ERR RESUME NEXT will ignore all errors and skip to the next statement. The variables ERR and ERRL can be used to get the error number and line number. To clear the last error information use the statement ERR CLR.

ON ERR OFF or simply ERR OFF will disable error trapping and clear the last error information.

E = ERR returns the number of the error that occurred. This must be used to decode which error occurred (see Appendix B).

E = ERRL returns the line number that the error occurred.

The RESUME statement is used to return from an error trap subroutine. The previous error number and line number will be cleared (set to 0).

If an error occurs in the subroutine before RESUME is executed, the error message will be displayed and the program will stop. This is to avoid continuous looping through the subroutine.

EXAMPLE:

```
10 ON ERR RESUME NEXT         :'IGNORE ALL ERRORS
20 PRINT 10/INT(RND(-TI)*10)  :'POSSIBLE DIVIDE BY ZERO
30 IF ERR > 0 THEN PRINT"ERROR OCCURRED":END :ELSE GOTO20
```

ON KEY

PURPOSE:

To redirect program execution when there is keyboard activity.

SYNTAX:

ON KEY GOSUB *line#*
[ON] KEY OFF

DESCRIPTION:

ON KEY allows the programmer to act on a key press (except run/stop-restore and control keys) from anywhere in the program. This gives freedom to the programmer to focus on other tasks while still acting on keyboard input.

line# is the first line number of the subroutine to GOSUB. If line# is not in the program, an UNDEF'D STATEMENT will occur.

To retrieve the ASCII value of the key that was struck use the KEY function: K = KEY(0). See Appendix C for ASCII characters and values.

While in the key trapping subroutine key events are paused to avoid multiple calls at the same time. In this case, the input will go into the keyboard buffer for the next iteration of the subroutine.

Use the RETURN statement as you would with any GOSUB to continue executing statements from where it was called.

ON KEY OFF or simply KEY OFF will turn off key trapping.

EXAMPLE:

```
0 '***ACCEPT KEYS WHILE MAIN LOOP RUNS***
10 ON KEY GOSUB 1000
.
.(main program loop here)
.
999 PRINT"DONE.":END
1000 CURSOR OFF
1005 IF KEY=13 THEN KEYOFF:RETURN 999
1010 PRINT KEY$;:CURSOR ON:RETURN
```

PAINT

PURPOSE:

Fills in a certain area of a bitmap screen with a specified color.

SYNTAX:

PAINT *x, y [,color]*
PAINT *x1, y1* **TO** *x2, y2 [,plotType] [,color*]

DESCRIPTION:

PAINT is used to fill an area on a bitmap screen (SCREEN 5) with plotted pixels. The first syntax uses a painting algorithm (flood fill) that plots pixels inside boundaries made by other contiguous pixels that are already plotted. The painting begins at the specified point and continues plotting until the entire enclosed area is filled. The area to be painted must be entirely enclosed with pixels. Any opening of even one pixel in size will result in painting outside the intended area.

x, y define the coordinates where the filling in of an object on the screen will begin. In hires mode the maximum values are x=0-319, y=0-199, but in multicolor mode, x=0-159, y=0-99. Any values out of range in the mode selected will result in an ILLEGAL COORDINATE ERROR.

The second syntax paints all pixels inside the rectangle defined by *x1*, *y1* (top-left) and *x2*, *y2* (bottom-right) using the specified *plotType* (0-3: 0=erase dot, 1=plot dot, 2=toggle dot, 3=none).

color (0-15 in hi-resolution mode, 1-3 in multicolor mode), is an optional parameter to select the paint color. If omitted then the last color selected by any graphics statement will be used. Refer to the MAPCOL statement for details about the available colors for both hires and multicolor bitmap modes. Refer to the COLOR statement for a list of available colors.

As with all graphics statements, no check is made to ensure the current screen is a bitmap screen. This allows making changes to the bitmap without being observed by the user. Showing the bitmap can be done using the SCREEN statement.

EXAMPLE:

```
0 COLOR ,0,0                  :'SET BORDER AND BACKGROUND COLORS
10 MAPCOL 1,5,7               :'SET MULTICOLOR BITMAP COLORS
20 SCREEN CLR 5,1             :'SHOW MULTICOLOR BITMAP AND CLEAR IT
30 CIRCLE 160,100,30,30,,1,2  :'DRAW A GREEN CIRCLE
40 PAINT 160,100,3            :'PAINT CIRCLE YELLOW STARTING AT CENTER
50 WAIT 200                   :'WAIT 200 JIFFIES BEFORE PRG ENDS
```

PEN()

PURPOSE:

Return the current coordinates of the light pen.

SYNTAX:

P = **PEN(***axis***)**

DESCRIPTION:

The C64 supports a single light pen device that can only be plugged into controller Port 1. When the light pen is held up to the screen the coordinates of the pen are captured and the button is pressed. This scan occurs 60 times per second (every screen frame). The PEN function returns the last captured value for the selected *axis* by taking the average value from 4 consecutive frames (4/60 or 1/15 of a second). This improves accuracy and compensates for an inaccurate light pen and a shaky hand.

The coordinate system is based on the raster scan line. Only 200 lines (50-249) are part of the visible display. They do not align to the bitmap or sprite coordinate system so a small adjustment is needed in a program to bring them into alignment. Also, the horizontal coordinate is only accurate every second dot position. This is due to the hardware limitation of representing a nine bit value with only eight bits, thus the horizontal axis value is an approximation with the visible area being about 42-362 (320 hires pixels). Results may vary between light pen manufacturers.

axis (0 to 2) is the PEN information to read:
0 = Flag indicating the pen is currently on the screen, 0=No, otherwise Yes
1 = Last captured raster x-axis of the pen
2 = Last captured raster y-axis of the pen

EXAMPLE:

```
0 DIM X,Y:COLOR 14,6,14:SCREENCLR
2 SPRITE0,1,1,0,13:POKE832TO832+63,255
10 PRINT CHR$(19);"WAITING FOR LIGHT PEN ACTION..."
20 IF PEN(0)=0 THEN 20 :'WAIT FOR PEN TO TOUCH SCREEN
30 X=PEN(1):Y=PEN(2)
35 SYS $E9FF,0,1 :'CLEAR LINE 1
40 IF Y<8 OR X<18 THEN PRINT"INVALID";X;Y;:GOTO 10
45 PRINT "LAST SCREEN POSITION: ";X;Y;
50 MOVE 0 TO X-18, Y-8,100
55 GOTO 10
```

PLAY

PURPOSE:

To play musical notes on any one of the three available SID voices.

SYNTAX:

PLAY *S$*
PLAY OFF

DESCRIPTION:

The PLAY statement uses voices 1 to play real musical notes at nine different octaves in foreground or background mode. If the main volume is currently set to 0 then it will automatically be set to the maximum of 15.

S$ is the string of notes to play. Invalid characters will be ignored.

There are 7 unique notes (A to G) in each octave. Sharps (# or +) and Flats (-) can be used to play the notes between notes like the black keys on a piano. Below is the table of symbols available to play music:

!	Play notes in foreground (must be first character in string)
@	Start over from beginning (should be last character in string)
A-G[#|+|-][*n*]	Musical note to play with optional length *n jiffies*
L*n*	Length *n* (0-99, default 30) in jiffies for all notes
O*n*	Octave *n* (0-8, default 4) for all notes
< | >	Decrease or increase the current octave number by 1
P[*n*]	Pause play for length *n (0-99, default 30) jiffies*
S[*n*]	Sustain volume *n* (0-15, default 15) for all notes
V*n*	Voice *n* (1-3, default 1) for all notes
W*n*	Waveform *n* (0-8, default 4 @ 50% duty) for all notes Refer to the WAVE statement for available waveforms

PLAY OFF will abort notes that are playing in background mode (if any).

EXAMPLE:

```
PLAY "!V3W1L20 A-A A#10 B-CC# P DD#EFF#GG# > A-AA#10B-CC# P60 DD#EFF#GG#"
```

PLAY SPRITE

PURPOSE:

To define and control animation for any of the eight available sprites.

SYNTAX:

PLAY SPRITE *spriteNum*, *startPtr*, *endPtr*, *speed*
PLAY SPRITE *spriteNum* **OFF**
PLAY SPRITE OFF

DESCRIPTION:

Sprite animation is achieved by changing a sprite's data pointer to another image and some defined rate. Each consecutive image promotes the illusion of motion.

spriteNum (0-7) is the sprite number to animate.

startPtr (0-255) is the sprites starting data pointer (first image).

endPtr (0-255) is the sprite's ending data pointer (last image).

The start and end pointers represent a range of consecutive integers where *startPtr* < *endPtr,* however this is not enforced and will cause incorrect images to display if violated. The pointer range should refer to sprite images that when played from start to end show a seamless animation. After the last image is displayed the first image is displayed again. This occurs continuously until turned off. See page 64 for details on selecting memory locations for sprite data.

speed (0-255 jiffies) is the time for each image to display.

PLAY SPRITE *spriteNum* OFF will stop animation for the given sprite.

PLAY SPRITE OFF will stop animation for all sprites.

EXAMPLE:

```
10 SPRITE 0,1,6
20 PLAY SPRITE 0, 200,206, 5
30 MOVE 0, 24,50 TO 300,200, 200
40 PLAY SPRITE 0 OFF
```

PLOT

PURPOSE:

Turns a pixel on or off on a bitmap screen.

SYNTAX:

PLOT *x, y [,plotType] [,color]*

DESCRIPTION:

The PLOT statement is used to change an individual pixel's state and color. It can set, clear or flip the pixel's on/off state at a specified coordinate. It can also be used to simply set the current plot coordinates without any change to the pixel. This is useful when using the DRAW statement.

x (0-319), *y* (0-199) in hi-resolution mode, *x* (0-159), *y* (0-99) in multicolor mode. *x, y* are the coordinates referencing a pixel on a bitmap screen. Any values out of range in the mode selected will result in an ILLEGAL COORDINATE ERROR.

plotType (0-3) determines how the dot will be plotted as follows:

PLOT TYPE	FUNCTION
0	Erase pixel
1	Plot pixel (default)
2	Flip pixel (on=off, off=on)
3	None, set current plot coordinate only – useful with DRAW statement

color (0-15 in hi-resolution mode, 1-3 in multicolor mode), sets the color for the pixel. See the MAPCOL statement for a list of the available colors for both modes.

EXAMPLE:

```
PLOT 160,100     :'PLOT CENTER OF HIRES SCREEN
PLOT 80,50,1,1   :'PLOT CENTER OF MULTICOLOR SCREEN WITH WHITE DOT
PLOT 0,0,0       :'TURN OFF PIXEL IN TOP LEFT CORNER
PLOT 319,199,2   :'FLIP BOTTOM RIGHT CORNER PIXEL'S CONDITION
```

POKE

PURPOSE:

To write a value to one or more memory addresses.

SYNTAX:

POKE *address1*, *value*
POKE *address1* **TO** *address2*, *value [,operation]*

DESCRIPTION:

POKE is the CBM BASIC statement to write a value to memory. MDBASIC has augmented this statement to write a value to multiple sequential addresses. This is many times faster than using a FOR/NEXT loop.

address1 (0-65535) is the start address of a memory to write.

address2 (0-65535) is the end address of the memory to write.

value (0-255) is the value to write at the specified address(s).

operation (0=none/set, 1=AND, 2=OR, 3=EOR) is the optional binary calculation to perform while writing the value. When omitted, the default *operation* is 0 which simply writes the *value* to memory. The *operation* is performed on each byte currently stored at each memory address in the range of addresses (inclusive).

EXAMPLE:

```
POKE 1024,1          :'WRITE THE VALUE 1 TO MEM ADDRESS 1024
POKE 832 TO 896,0    :'WRITE THE VALUE 0 TO MEM ADDRESSES 832-896
POKE $0400 TO $07E7, %10000000, 3 :'FLIP BIT 7 OF ALL CHARS ON SCREEN

0 '***BLINKING TEXT EXAMPLE***
10 CURSOR 29,0 : PRINT "BLINKING";
20 POKE 1024+29 TO 1024+39, %10000000, 3
30 WAIT 20
40 GOTO 20
```

POT()

PURPOSE:

Return the current condition of any of the four game paddles.

SYNTAX:

P = **POT(***paddleNum***)**

DESCRIPTION:

The state of the paddle controllers are read using this function. POT is short for potentiometer which is an electronics term for the variable resistor used inside the controller.

paddleNum can range from 1-4. Each port controls two paddles.

Controller Port 1 monitors paddles 1 & 2
Controller Port 2 monitors paddles 3 & 4.

The position of the paddle dial is returned as a value between 0 and 255 (8 bits). If the button is pressed, a value of 256 is added (bit 9). You can use the AND operation to focus on either the dial position or the button state. Refer to the example below.

EXAMPLE:

```
0 P1 = RND(-TI)              :'INITIALIZE RANDOM NUMBER GENERATOR
10 SPRITE 0,1,1,0,13         :'TURN ON SPRITE 0, WHITE, DATA PTR 13
15 POKE 832 TO 832+63,255    :'FILL SPRITE DATA WITH ALL BITS ON
20 MOVE 0,24,50              :'PUT SPRITE IN VISIBLE AREA OF SCREEN
25 P1 = POT(1)               :'CAPTURE POT#1'S POSITION
30 MOVE 0, (P1 AND 255)+24   :'SPRITE 0 X COORDINATE USES POT 1 POSITION
35 'WHEN BUTTON PRESSED CHANGE SPRITE COLOR RANDOMLY, SHORT PAUSE
40 IF P1 > 255 THEN SPRITE0,,RND(1)*15: WAIT 10
50 GOTO 25
```

PTR()

PURPOSE:

Returns the memory address of a variable’s value or pointer.

SYNTAX:

P = **PTR(***variable***)**

DESCRIPTION:

PTR returns the memory address of a *variable*'s descriptor.

Non-array variable values use five bytes regardless of data type. Strings will only use the first 3 bytes, integers the first 2 bytes and floats use all 5 bytes.

Array variables data size depends on the data type. Strings use 3 bytes, integers use 2 bytes and floats use 5 bytes.

A floating point value will always use five bytes to store the value in 32-bit single-precision floating point format (IEEE 754).

An integer will only use two bytes in big-endian format (most significant byte first).

A string variable will use the first byte for its length, and the second and third bytes for the memory address in little-endian format (least significant byte first) of the first byte in the string. The actual string text that is pointed to is located either in the part of the BASIC program where the string is first assigned a value, or in the string text storage area pointed to by location 51-52 ($33-$34).

EXAMPLE:

```
P1 = PTR(N)     :'P1 IS THE ADDRESS OF THE 5-BYTE FLOATING-POINT VALUE
P2 = PTR(I%)    :'P2 IS THE ADDRESS OF THE 5-BYTE INTEGER VALUE
P3 = PTR(I%(0)) :'P3 IS THE ADDRESS OF THE FIRST 2-BYTE VALUE IN ARRAY I%

0 '***STRING DESCRIPTOR EXAMPLE PRG***
5 DIM I,P,L,A
10 S$="TEST" : P=PTR(S$)
20 L=PEEK(P) : A=(256*PEEK(P+2))+PEEK(P+1)
30 PRINT"STR LEN:";L
40 PRINT"STR ADDRESS:";A;CHR$(13);"VALUE: ";
50 FOR I=A TO A+L-1 : PRINT CHR$(PEEK(I)); : NEXT
```

PULSE

PURPOSE:

Sets the pulse value for any of the three voices.

SYNTAX:

PULSE *voice, duty*

DESCRIPTION:

PULSE sets the pulse width of any voice. This feature only works when a pulse waveform has been selected, which the output of the signal is a rectangular wave.

voice (1-3) selects which voice to be affected.

duty (0.0 to 100.0) determines the duty cycle percentile of time that the rectangular wave will stay at the high part of the cycle. Changing the pulse width will vastly change the sound created when using the pulse waveform.

In order for this statement to affect the sound, one of the pulse waveforms (4,5,6 or 7) must be selected for the corresponding voice by using the WAVE statement. Refer to the WAVE statement for details on selecting a waveform.

EXAMPLE:

```
0 'PULSE 0 TO 100
5 SCREENCLR:PRINT"PULSE WIDTH: 0";
10 VOICECLR:VOL15
15 VOICE1,900
20 ENVELOPE1,0,0,15,0
25 WAVE1,4,1
30 FORI=0TO100:PULSE1,I
35 CURSOR12,0:PRINTI;:WAIT5
36 NEXT
50 WAVE1,4,0:WAIT20
55 VOICECLR
```

RENUM

PURPOSE:

To renumber a BASIC program with a specific line increment.

SYNTAX:

RENUM
RENUM *[start] [,increment]*

DESCRIPTION:

The RENUM command is used to renumber all BASIC program lines while maintaining line number references in statements like GOTO, GOSUB, etc. Renumbering is typically done to add numerical separation between each program line to make it easy to insert additional lines. Consistent line numbering also makes a program look neat and easy to follow. When merging two programs together using MERGE command, both programs should be renumbered so that the line numbers do not overlap. Refer to the MERGE command for more details.

start is the optional (default 0) new starting line number for the program.

increment is the optional (default 10) number of lines between every line.

If RENUM has no operands, then the default value of 10 will be used for the start line number and increment.

RENUM changes all line numbers that have statements referring to line numbers. Below is the list of such kewords:

THEN, ELSE, GOTO, GOSUB, RESTORE, RETURN, RESUME, RUN, TRACE, DELETE

If RENUM comes across a reference to a line number that is not in the program, the number is replaced with 65535, and the line number that the referencing statement is on is listed.

NOTE: RENUM will not display any messages while processing which can take several seconds depending on the size of the program and the number of statements that reference line numbers. Expect to wait an average of 5 seconds per 100 lines of code.

EXAMPLE:

```
RENUM           :'PROGRAM STARTS WITH LINE 10 WITH 10 INCREMENTS
RENUM 1000,10   :'PROGRAM STARTS WITH LINE 1000 WITH 10 INCREMENTS
RENUM 100       :'PROGRAM STARTS WITH LINE 100 WITH 100 INCREMENTS
```

RESTORE

PURPOSE:

Resets the data pointer to either the first or specific data statement.

SYNTAX:

RESTORE
RESTORE *[lineNum]*

DESCRIPTION:

RESTORE is an existing CBM BASIC statement which resets the data pointer to the first data line in the program. It has been augmented to support restoring to an optionally specified line number in the program.

When all DATA in a program has been consumed by the READ statement, any attempt to READ more data will cause an OUT OF DATA ERROR. RESTORE allows re-reading of data at any time, at any DATA statement, as many times as necessary.

RESTORE without a line number will set the next READ to the first DATA statement in the program. No error occurs if the BASIC program does not contain any DATA statements. This is the CBM BASIC implementation.

RESTORE *lineNum* will set the line number for the next READ statement to get its DATA. If *lineNum* is not a valid line number in the program, an UNDEF'D STATEMENT ERROR will occur. If *lineNum* does not have a DATA statement, the next READ statement will search from that point to the end of the program for the next DATA line. If no DATA line is found, then the next READ statement will cause an OUT OF DATA ERROR.

EXAMPLE:

```
RESTORE         :'SETS DATA POINTER AT FIRST LINE IN PROGRAM
RESTORE 100     :'DATA IS FOUND STARTING AT LINE 100
```

RESUME

PURPOSE:

To return from an error handling subroutine previously defined by the ON ERR GOTO statement.

SYNTAX:

RESUME
RESUME NEXT
RESUME *[lineNum]*

DESCRIPTION:

RESUME is used in conjunction with the ON ERR GOTO statement to return from an error handling subroutine. The location to return is described below.

RESUME used alone will return program execution to the statement that caused the error.

RESUME NEXT returns execution to the statement immediately following the statement that caused the error.

RESUME *lineNum* redirects the program execution to any line in the program. Resuming to an invalid *lineNum* will cause an UNDEF'D STATEMENT ERROR.

Regardless of how resuming is done, the previous error number is set to 0 and the previous error line number is set to -1 (ERR CLR) since it was handled.

Error trapping is turned off automatically when entering the error handling subroutine to avoid a continuous call to the subroutine which would eventually lead to a STACK OVERFLOW ERROR. RESUME will re-enable the last used error handling definition set by ON ERR RESUME *line#*. Therefore, it is possible to nest multiple error handlers by carefully setting & resetting the definition.

RESUME is the only statement that cannot be trapped by ON ERR GOTO. If executed without an error trapped by ON ERR GOTO line# then CAN'T RESUME ERROR will occur. Resuming to an invalid *line#* will cause an UNDEF'D STATEMENT ERROR.

EXAMPLE:

```
10 ON ERR GOTO 1000
…(main program here)…
999 END
1000 PRINT"ERR:";ERR;"LINE:";ERRL
1001 RESUME NEXT:REM SKIP OVER STATEMENT THAT CAUSED THE ERROR
```

RETURN

PURPOSE:

To return from a subroutine called by the GOSUB statement.

SYNTAX:

RETURN
RETURN *[lineNum]*

DESCRIPTION:

RETURN is a CBM BASIC statement used to return to the statement after the GOSUB statement that called the function. It has been augmented by MDBASIC to allow the safe redirection of the program to a specified line number instead of the statement that followed the GOSUB statement that made the call.

RETURN without a line number returns to the statement following the GOSUB statement that made the call. This is the standard CBM-BASIC behavior.

RETURN *lineNum* will abort the GOSUB call by discarding the five bytes of stack information pertaining to the call then perform a GOTO with the line number provided. This is the MDBASIC augmentation of the RETURN statement.

Returning to a different point in the program breaks the rules of control-flow programming, however, traditional BASIC is not a control-flow language and is dependent on line numbers on every line of program text.

Using the RETURN statement without previously executing a GOSUB statement will cause RETURN WITHOUT GOSUB ERROR.

EXAMPLE:

```
10 A=1: PRINT"THIS IS A TEST"
20 GOSUB 100:PRINT"RETURNED BACK"
25 PRINT"*********"
30 PRINT"DONE"
40 END
100 PRINT"*SUBROUTINE*"
110 IF A > 0 THEN RETURN30
120 RETURN
```

ROUND()

PURPOSE:

Round a floating point number to a specified precision.

SYNTAX:

N = **ROUND(***float [,precision]***)**

DESCRIPTION:

ROUND is a numeric function that returns the given value rounded to a specified number of decimal places. ROUND is useful for correcting inaccuracies with single-precision, floating-point numbers.

float is a floating-point variable (or constant) of the number to round.

precision (optional, signed integer -9 to 9 default 0) is the number of digits left (negative) or right (positive) of the decimal point. If no *precision* is specified the default is 0 which returns the nearest whole number value. When *precision* is negative, the whole number portion will be rounded to the nearest number's place. For example: -1 is tens place, -2 is hundreds place, etc.

There are limitations to accuracy that can produce incorrect rounding results. The 32-bit binary representation uses the same bits for the whole number portion and the fractional portion. A smaller whole number value can have more fractional digits and vise-versa. Accuracy problems can occur with numbers that have a large whole number and fractional portion. However, values that are between -999999.000 and 999999.000 rounded with *precision* between 0 and 3 should be accurate and cover most use cases.

EXAMPLE:

```
PRINT ROUND(5.0125),ROUND(5.0125,0),ROUND(5.0125,2),ROUND(5.0125,3)
5         5         5.01      5.013

PRINT ROUND(125.23),ROUND(125.23,-1),ROUND(125.23,-2),ROUND(125.23,-3)
125       130       100       0
```

RUN

PURPOSE:

Execute a program, or load a program into memory and execute it.

SYNTAX:

RUN
RUN [*line#]*
RUN [@] *filename$ [,device#]*

DESCRIPTION:

RUN works the same as the Commodore version with one addition, which allows a program to be automatically loaded into memory & run form direct mode or program mode. This statement is useful when linking execution of two separate programs.

line# is the line that the program will start execution. If no line number is selected, the program starts execution at the first line.

filename$ is a string of characters that represent the name of the program that will be loaded into memory and automatically executed. The previous program in memory will be lost as well as any variable values.

device# (0-31, default 1) is the device number from where the merging program will load. Device 1 is for tape, devices 8 to 12 are for disks.

RUN will always perform the following initializations:
1. Clear all variables (CLR)
2. Turn off error trapping (ERR OFF) and key trapping (KEY OFF)
3. Turn off the printing of Kernal and BASIC messages

RUN@ will skip these additional initializations:
4. Clear the keyboard buffer (KEY CLR)
5. Close all open files (CLOSE FILES) and set default devices & I/O channel
6. Turn off all MDBASIC background processes (PLAY OFF, PLAY SPRITE OFF)
7. Turn off all sprites (SPRITE OFF)
8. Clear last error info (ERR CLR)
9. Turn off sound by clearing all SID registers (VOICE CLR)

RUN will not affect the current state of the video matrix. Any text on the screen, color settings and graphics mode will remain the same. Each program should initialize these settings if needed.

EXAMPLE:

```
RUN 100        :'RUN CURRENT PROGRAM IN MEMORY STARTING AT LINE 100
RUN"PROG1"     :'LOAD & RUN "PROG1" FROM TAPE
RUN@"PROG2",8  :'LOAD & RUN "PROG2" FROM DISK WITHOUT ADDITIONAL INITIALIZATIONS
```

SAVE

PURPOSE:

To write content of RAM to a device for later retrieval with LOAD.

SYNTAX:

SAVE
SAVE *[filename$] [,device] [,secondary]*
SAVE *address1, address2 [,filename$] [,device] [,secondary]*

DESCRIPTION:

SAVE is a CBM BASIC command that has been augmented to support saving a contiguous section of RAM. This type of save is commonly referred to as a "binary save". The first syntax listed above is the CBM BASIC syntax which is from the start address of BASIC program text ($0800) to the address of the end of the current BASIC program text. The second syntax is the augmented SAVE which supports a custom address range.

Some examples that make use of a binary SAVE are:
1. Save sprite data to a file for quick retrieval with LOAD
2. Save a BASIC program with an appended assembly language subroutine.
3. Save the video matrix (text screen)

NOTE: To save the screen with color, the bitmap or a redefined character set use the associated secondary address as described in Appendix A.

address1 (0-65535) is the start address in RAM for the first byte to save.

address2 (0-65535) is the end address in RAM for the last byte to save.

filename$ is the name of the file and should not exceed 16 characters. Tape devices do not require a name for the file and can be an empty string.

device (0-11, default 1) is the device number to write the bytes.
0=Keyboard, 1=Dataset, 2=RS-232/User Port, 3=Screen, 4-5=Printer, 8-11=Disk

secondary (0-31 for serial devices, 0-127 for other devices, default 0) is the secondary address value with meaning that is specific to the device.

EXAMPLE:

```
SAVE "MY_BAS_PRG", 8             :'SAVE CURRENT BASIC PROGRAM TO DISK
SAVE $C000, $CFFF, "ML_PRG", 8   :'SAVE 4K BYTES IN HIRAM TO DISK
SAVE 1024, 2023, "SCREEN",  8    :'SAVE TEXT SCREEN TO DISK
```

SCREEN

PURPOSE:

To control the screen display including page selection, color mode and fine scrolling.

SYNTAX:

SCREEN ON | **OFF**
SCREEN CLR
SCREEN ***[*****CLR*****]*** *[page] [,colorMode] [,offsetX] [,offsetY]*

DESCRIPTION:

The screen statement controls the display of the entire screen. There are 5 available pages which share the same color RAM. The screen can be turned ON or OFF which can improve CPU performance. The screen borders can be enlarged to hide the screen offset and incoming characters being added to the screen when fine-scrolling text. It can also hide an enlarged sprite so that it does not stick out of the border at its zero axis. The screen offset can also be used to produce a full-screen shaking effect.

SCREEN ON and SCREEN OFF enables/disables the screen output. The color of the entire screen will be the color of the border. This can improve CPU performance by reducing traffic on the data bus. This can allow files to be correctly loaded from old 1540 disk drives.

SCREEN CLR will clear the current screen. If the screen is a bitmap then the plotted dots are removed and the background color is set according to the values set by the last MAPCOL statement (for normal/hires mode) or the COLOR statement (for multicolor mode).

page (0-5) selects the page to display. Pages 0-4 are 1K pages in text mode. On first use, pages 1,2,3 & 4 require initializing the character dot data (DESIGN NEW or LOAD from file) which is shared by these four pages. Page 5 is the 8K bitmap page. Each page will need to be cleared on first use. This can be done flicker-free by preceding the *page* parameter with the CLR keyword.

PAGE	SCREEN RAM	SPRITE POINTERS	AVAILABLE IMAGE RAM	SPRITE DATA INDEXES
0	$0400-$07E7	$07F7-$07FF	$0340-$03FF, $0800-$3FBF	13-15,32-255
1	$C000-$C3E7	$C3F7-$C3FF	$C400-$CFFF	16-63
2	$C400-$C7E7	$C7F7-$C7FF	$C000-$C3FF, $C800-$CFFF	0-15, 32-63
3	$C800-$CBE7	$CBF7-$CBFF	$C000-$C7FF, $CC00-$CFFF	0-31, 48-63
4	$CC00-$CFE7	$CFF7-$CFFF	$C000-$CBBF	0-47
5	$C800-$CBE7	$CBF7-$CBFF	$C000-$C7FF, $CC00-$CFFF	0-31, 48-63

NOTE: Page 3 is used by the bitmap screen. Page 4 is used by the RS-232 I/O.

colorMode (0-2, default 0) is used to select the color scheme for all screens.

MODE	DESCRIPTION	TEXT	BITMAP
0	Normal (default)	8x8	320x200
1	Multicolor	4x8	160x200
2	Extended Background Color	8x8	n/a

Normal color mode is the default color mode for both text and bitmap screens. For text mode, each character can only have one color (0-15) for all dots that define the font in the 8 x 8 matrix. All characters on the screen share the same background color (0-15).

A normal color (hires) bitmap screen can have each pixel with one of two states, on or off. The pixels that occupy the same 8 x 8 square will share the same color (0-15) based on the pixel state. Screen 3 is used by the bitmap screen for this purpose. The color of each 8 x 8 square corresponds to a character on the video matrix. The high-nybble determines the color for pixels that are off. The low-nybble determines the color for pixels that are on.

Multicolor mode is used by both text and bitmap screens. This mode increases the number of colors available for each character by reducing the number of pixels used to display it. Each pixel is represented by 2 bits. The bit pattern determines the color source.

The COLOR statement is used to set all colors for text screens. The background color, *cc1* and *cc2* are common for all characters on the screen. Any change is visible immediately. The table below lists the four 2-bit patterns and the associated color source:

PATTERN	COLOR	SOURCE	AVAILABLE COLORS
00	background color	background color reg 0	0-15
01	*cc1*	background color reg 1	0-15
10	*cc2*	background color reg 2	0-15
11	foreground color	color RAM ($D800-$BDFF) first 3 bits, colors 0-7	0-7 select normal color mode, 8-15 select multicolor of 0-7

If a character is printed to the screen using colors 0-7, the character is displayed in multicolor mode (4 x 8). If it is printed using colors 8-15, the character is displayed in normal mode (8 x 8). This allows multicolor & hi-resolution characters to be displayed on the same screen at the same time, at expense of loosing colors 0-7 for hires mode.

If a character is not available in the scan codes 0-63 then the DESIGN statement can be used to define a custom shape. Entire character sets can also be loaded from a file into memory at $F000-$FFFF. Multicolor character fonts are typically made up of two characters placed side-by-side with the shift key characters used for the second half of each character.

Extended background color mode enables multiple background colors for each text character. This mode increases the number of background colors displayed by reducing the number of characters that can be shown on the screen. The only displayable characters are the characters with codes from 0-63 (5-bit value). The upper 2 bits are used to select the source of the color. Codes 64-255 select a different background color but still display the same character of codes 0-63. The COLOR statement is used to set the colors to be selectable by each bit pair.

PATTERN	COLOR	CODES	SOURCE	AVAILABLE COLORS
00	background color	0-63	background color reg 0	0-15
01	*cc1*	64-127	background color reg 1	0-15
10	*cc2*	128-191	background color reg 2	0-15
11	*cc3*	192-255	background color reg 3	0-15

Extended background color mode can be used to produce a window-like effect on sections of text because a different background color makes text stand out from surrounding text having another background color. The COLOR statement can be used to change the background colors of these windows instantly to highlight a particular section of text on the screen or produce a flashing effect. Extended background color mode cannot be combined with Multicolor Text or Bitmap modes. Enabling one disables the other.

offsetX (0-15, default 8) controls the horizontal foreground offset and border size. Values 0-7 set the offset with 38 visible columns. Values 8-15 set the same offset but with 40 visible columns.

offsetY (0-15, default 11) controls the vertical foreground offset and border size. Values 0-7 set the offset with 24 visible rows. Values 8-15 set the same offset but with 25 visible rows.

EXAMPLE:

```
SCREEN OFF:WAIT60:SCREEN ON    :'TURN SCREEN OUTPUT OFF FOR 60 JIFFIES
SCREEN,,0                      :'38 COLUMNS (EXPANDED SIDE BORDERS)
SCREEN 0,0,8,11                :'NORMAL C64 SCREEN
SCREEN CLR 5,0                 :'INIT AND SHOW HIRES BITMAP
DESIGN NEW:SCREEN CLR 1        :'INITIALIZE FONT AND SWITCH TO PAGE 1 CLEARED
SCREEN ,2                      :'ENABLE EXT BKGND COLOR MODE
```

SCROLL

PURPOSE:

To move a section of text in a specified direction with optional wrapping.

SYNTAX:

SCROLL *x1, y1* **TO** *x2, y2 [,direction] [,wrap]*

DESCRIPTION:

SCROLL allows any section of text on the screen to be scrolled up, down, left, or right. The section is moved in the specified direction by one character per statement execution. SCROLL and SCREEN statements can be used together in a loop to achieve fine scrolling text. See the example at the end of Appendix E.

x1 (0-39), *y1* (0-24) define the coordinates of the first character, on a text screen, to scroll. This is the upper left corner of the display that will be scrolling.

x2 (0-39), *y2* (0-24) define the coordinates of the last character, on a text screen, to scroll. This is the lower right corner of the display that will be scrolling.

direction (0-3, default=0) defines the direction of scroll as follows:

DIRECTION	MOVEMENT
0	up (default)
1	down
2	left
3	right

wrap (0=truncate, 1=wrap, default 0) specifies if the text scrolled off the screen is to wrap around to the other side. For example, scrolling up would wrap the top most line to the bottom. If wrap=0 then spaces will fill the bottom.

EXAMPLE:

```
SCROLL 0,0 TO 39,24   :'SCROLL WHOLE SCREEN UP, NO WRAPPING
SCROLL 0,0 TO 39,0,3  :'SCROLL TOP LINE RIGHT, WITH WRAPPING
```

SERIAL

PURPOSE:

To send and receive data through the RS-232 port.

SYNTAX:

SERIAL OPEN *[baudrate] [,databits] [,stopbits] [,duplex] [,parity] [,handshake]*
SERIAL ***[*****WAIT** *[timeout]****]*** **READ** *variable [***TO** *sentinel]*
SERIAL PRINT *[expression]*
SERIAL CLOSE

DESCRIPTION:

The SERIAL statement is a multi-functional command used to execute multiple operations on the RS-232 port. SERIAL must be followed by of of the keywords OPEN, READ, PRINT and CLOSE. The benefits of using SERIAL is speeding up the read process to keep up with higher baud rates. Also, SERIAL will not cause variable data loss or BASIC memory reduction as occurs with opening device 2 with OPEN. Also you can use these statements in direct mode if needed.

SERIAL OPEN is used to open the RS-232 channel for communication. If the port is already open the FILE OPEN ERROR will result.

baudrate (optional, default 1200) is the speed for bit transmission. The value must be 50, 75, 110, 134, 150, 300, 600, 1200, 1800 or 2400 otherwise an ILLEGAL QUANTITY ERROR will result.

databits (optional, 5, 6, 7 or 8, default 8) is the number of bits that make up the packet for one "word" of data. Some data, like ASCII characters do not need all 8 bits so it is better to reduce the size to increase overall throughput.

stopbits (optional, 0 or 1, default 0) is the number of bits used to provide markers in the transmission to demark the end of a packet of data (byte).

duplex (optional, 0 = full (default), 1 = half) controls the synchronization of data flow. Full duplex allows simultaneous send & receive transmission (like a telephone). Half duplex is one direction at a time (like a walkie talkie).

parity (optional, 0-4, default 0) controls how data packet errors are detected.

PARITY	DESCRIPTION
0	No Parity Generated or Received
1	Odd Parity Transmitted and Received
2	Even Parity Transmitted and Received
3	Mark Parity Transmitted and Received
4	Space Parity Transmitted and Received

handshake (0 = 3-Line (default), 1 = X-Line) is the signal (CTS/RTS) control of the transmission of data to help prevent data loss between a fast sender and a slow receiver.

SERIAL WAIT will suspend the program until the next byte of data is read. The time to wait is infinite unless a *timeout* is provided. When reading data into strings, each byte received resets the *timeout*.

timeout (optional, 0-65535 jiffies) indicates how many jiffies to wait for the next byte in the buffer before ending the read. If *timeout* is 0 or omitted then the wait will be infinite.

variable is the variable that will receive the result of the read. Numeric data types (float or integer) will only capture one byte at a time. Your program must have a loop to read each byte which is slower and may result in buffer overrun at higher baud rates. When the *variable* is a string, multiple bytes can be read in the same READ statement which is much faster. Note that binary data may have zero-byte values included in the string. If the amount of data exceeds 255 bytes then the string is returned, thus another read must be used to get more data.

sentinel is the byte which will stop the read and return the result which includes the *sentinel* byte itself. If the variable is numeric, all bytes up to the sentinel are discarded (skipped). If the *variable* is a string and the results reaches 255 bytes or the *timeout* is reached then the data is returned regardless of the presence of the *sentinel*. In this case an empty string, or a zero numeric value depending on the data type of the *variable*, will be returned. You can use the CBM BASIC variable ST to determine the status.

SERIAL READ will read a maximum of 255 bytes or up to an optionally specified stopping byte (sentinel), whichever condition comes first. The first read after opening the port will adjust for the start bit automatically.

SERIAL PRINT can be used to send data. The syntax is exactly like any PRINT statement. You can end the expression with a semicolon to prevent sending a carriage return character.

SERIAL CLOSE is used to finalize the communication on the RS-232 port. You must use CLOSE to change the parameters of the opened channel. Like CBM BASIC’s CLOSE statement, SERIAL CLOSE will never cause an error even if it wasn’t opened.

The ST variable can be used to determine the status of the last SERIAL READ. Be aware that the value of the ST variable will be cleared after reading its value so you may need to capture the value in your own variable if you need to refer to it more than once. See Appendix D for details of the status code.

See Appendix F for a more detailed example.

EXAMPLE:

```
SERIAL OPEN               :'OPEN RS-232 CHANNEL WITH USE DEFAULTS
SERIAL WAIT 300 READ S$   :'READ A STRING OF BYTES WITH 5 SEC TIMEOUT
SERIAL PRINT A$;          :'WRITE A STRING WITHOUT CARRIAGE RETURN
SERIAL CLOSE              :'CLOSE THE CHANNEL
```

SPRITE

PURPOSE:

To set/change the configuration of a sprite including the visibility, color & color mode, foreground priority, data index and expansion size for the image.

SYNTAX:

SPRITE *spriteNum, [visible] [,color] [,multi] [,index] [,priority] [,size]*
SPRITE *spriteNum* **EXPAND** *size*
SPRITE *spriteNum* **DATA** *index*
SPRITE ON | OFF
SPRITE EXPAND *size*

DESCRIPTION:

A sprite is a Movable Object Block (MOB) that is 24 bits wide and 21 bits tall consuming 63 bytes of memory per shape. Sprites have their own x & y coordinate system which differs from that of the bitmap screen. Any of the eight sprites can be moved anywhere on the screen, even under the border that surrounds the edge of the screen. Image display is hardware-driven so there is no need to erase and draw the image when the coordinates of a sprite changes. See the MOVE statement for more details.

Each SPRITE parameter can be skipped to target only the ones needing to change. Parameters can also be omitted to end the statement without specification.

spriteNum (0-7) selects the sprite that is affected.

visible (0 or 1) is a Boolean expression that turns the selected sprite on or off. Any other value will result in an ILLEGAL QUANTITY ERROR.

color (0-16) selects the color for the sprite with 16 being a glow effect.

multi (0 or 1) is a Boolean expression that selects the color mode (0=hires, 1=multicolor). If multicolor mode is selected, more colors can be used in the sprite, but the horizontal resolution is cut in half to facilitate this feature. Use the SPRCOL statement to set the colors *sc1* & *sc2* used by all sprites. The table below lists the four 2-bit patterns and the associated color:

PATTERN	COLOR
00	background color
01	*sc1*
10	sprite color
11	*sc2*

Designing a multicolor sprite is the same as a multicolor character when arranging the bit-pairs for the corresponding colors (See DESIGN statement).

index (0-255) is the index of the 64-byte data block that defines the shape of the sprite. The memory address can be calculated by the formula:

ADDRESS=(INDEX*64)+(BANK*16384)

BANK (0-3) is the 16K VIC-II base video memory bank. MDBASIC selects the memory bank depending on the active graphics mode selected.

MDBASIC MODE	BANK	MEMORY RANGE	COMMENTS
Standard Text	0	0-16383	1K SYS RAM, 1K VIDEO RAM, 14K BASIC RAM
n/a	1	16834-32767	16K BASIC RAM
n/a, Restricted	2	32768-49151	16K MDBASIC RAM, 8K CBM BASIC ROM
Bitmap Graphics & Custom Text	3	49152-65535	4K HIRAM, 4K DEVICE RAM, 16K KERNEL ROM 1K VIDEO RAM is at 51200-52223

priority (0 or 1) is a Boolean expression that determines the priority of the sprite graphics to foreground graphics. When *priority* is set to 0 (default) the sprite will appear over all bitmap graphics and text characters; a value of 1 will make the sprite appear underneath. If this operand is greater than 1, an ILLEGAL QUANTITY ERROR will result.

size (0 to 3) is the expansion sizing options for the sprite. A sprite's horizontal and vertical dimensions can be displayed twice the normal size. When an axis is expanded, each pixel is drawn twice along that axis, which increases the size of the image. The table below lists the possible values for this parameter:

SIZE	DESCRIPTION
0	Normal (no expansion)
1	Horizontal expansion only
2	Vertical expansion only
3	Horizontal and vertical expansion

Below is a list of alternate syntax for setting sprite parameters:

1. Setting only the *size* parameter:
SPRITE *spriteNum*,,,,,,*size*
SPRITE *spriteNum* EXPAND *size*

2. Setting only the image *index* parameter:
SPRITE *spriteNum*,,,,*index*
SPRITE *spriteNum* DATA *index*

3. Setting only the *visible* parameter for all sprites:
SPRITE ON
SPRITE OFF

4. Setting only the expansion *size* parameter for all sprites:
SPRITE EXPAND *size*

To design a sprite shape you need 63 bytes of data which is placed into memory starting at the address calculated using the formula described above. The data can be entered in DATA statements or loaded from a file.

MDBASIC comes with a sprite editor that is easy to use and will help develop the sprite shape needed in hires or multicolor mode. The following is an example of how a hires sprite is made and turned into data:

BINARY CODE	DATA
000011111111111111000000	15,255,192
001111111111111111110000	63,255,240
001111111111111111110000	63,255,240
011111000000000011111000	124,0,248
011100001000010000111000	112,132,56
111100011000011000111100	241,134,60
111100000000000000111100	240,0,60
111000000011000000011100	224,48,28
111000000011000000011110	224,48,30
111001100000000110011110	230,1,158
111000110000001100011110	227,3,30
111000011111111000011110	225,254,30
111100000000000000111110	240,0,62
111110000000000000111110	248,0,62
111111100000000011111110	254,0,254
111111110000000111111110	255,1,254
111111110000000111111100	255,1,1,252
111111110000000111111100	255,1,252
110000000000000000000110	192,0,6
100000000000000000000001	128,0,1
100000000000000000000001	128,0,1

Every three bytes makes one line of pixels. There are 21 lines in a sprite totaling 63 bytes. The data can easily be put into DATA statements or a sequential file for a loop to READ and POKE them into memory.

Since sprite memory is represented in 64-byte blocks you will need to add an additional byte to the end of your data if you have multiple consecutive images. This unused byte could be use to hold additional data about the sprite. For example, the color (4 bits), horizontal & vertical expansion (2 bits), visibility (1 bit) and priority (1 bit) could all be encoded the bits of the 64th byte which you can apply to the image after loading or reading the data.

EXAMPLE:

```
SPRITE 0,1,7,0,13  :'SPRITE 0 ON, YELLOW, NO MULTICOLOR, DATA POINTER 13
SPRITE 0,,,,14     :'CHANGE SPRITE 0'S POINTER TO 14
SPRITE 0 DATA 14   :'SAME AS ABOVE WITH ALT SYNTAX
SPRITE 1,,,,,,3    :'CHANGE SPRITE 1'S SIZE TO EXPAND WIDTH AND HEIGHT
SPRITE 1 EXPAND 3  :'SAME AS ABOVE WITH ALT SYNTAX
SPRITE 2,,,,,1     :'CHANGE SPRITE 2'S PRIORITY TO BE UNDER TEXT/GRAPHICS
SPRITE ON          :'TURN ON ALL SPRITES
SPRITE OFF         :'TURN OFF ALL SPRITES
SPRITE EXPAND 3    :'EXPAND ALL SPRITES HORIZONTAL AND VERTICAL
```

SPRCOL

PURPOSE:

To select a multicolor mode and set colors used by the foreground text, background text or any sprite that has multicolor mode enabled.

SYNTAX:

SPRCOL [*sc1] [,sc2] [,colorModeFlags]*

DESCRIPTION:

SPRCOL selects colors for sprites having multicolor mode enabled. Colors *sc1* & *sc2* define two additional colors that that can occupy the same 24 x 21 sprite. All sprites in multicolor mode will share these extended colors. The table below lists the four 2-bit patterns with the source of the color value:

PATTERN	COLOR
00	background color
01	*sc1*
10	sprite color
11	*sc2*

On system startup the default colors are *sc1*=4 (Purple) and *sc2*=0 (Black). To be able to see all the colors it is important to set the screen's background color to a different color than the colors selected.

colorModeFlags is an 8-bit value representing the eight available sprites. The bits are used as flags to turn on or off multicolor mode for each sprite. Multicolor mode can be individually enabled for a specific sprite using the SPRITE statement. This parameter offers an alternative to select the color mode for all eight sprites in one statement.

Designing a multicolor sprite is similar to designing a multicolor character when arranging bit-pairs to select the corresponding color (See DESIGN statement).

EXAMPLE:

```
SPRCOL 5,6     :'SELECTS COLORS FOR ALL SPRITES WITH MULTICOLOR MODE ENABLED
SPRCOL 1,2,255 :'SELECTS COLORS AND ENABLES MULTICOLOR MODE FOR ALL SPRITES
SPRCOL ,,0     :'TURN OFF MULTICOLOR MODE FOR ALL SPRITES
```

STOP

PURPOSE:

Pause a running BASIC program or enable/disable the STOP statement or key.

SYNTAX:

STOP
STOP ON | OFF
STOP KEY ON | OFF

DESCRIPTION:

The STOP statement is a CBM BASIC statement that was augmented to provide additional features. Normally the STOP statement is used to pause a running BASIC program for debugging purposes. The BREAK ERROR IN line# message is displayed and the programmer can interrogate system state (see VARS command).

STOP ON | OFF (default ON) allows the programmer to enable or disable the STOP statement. This is handy when the program is to run normally without interruption but leaving the various STOP statements in the code for future debugging. This statement would typically be at the top of a program that uses this debugging feature.

STOP KEY ON | OFF (default ON) allows the programmer to decided if the Stop key is allowed to stop the program. This allows the ability to capture the key press in a BASIC program and act on it accordingly. For example you may want to prompt the user to confirm they would like to exit the program and if so, do some final housekeeping activities to shut down gracefully.

When the Stop key is disabled the ASCII value of 3 will be returned when reading from the keyboard. The state of the stop key setting will persist even when returning to immediate mode or running another program from within a program, therefore, you should turn it back ON when done.

EXAMPLE:

```
10 STOP KEY OFF
20 KEY GET A:IF A=0 THEN20
30 IF A=3 THEN 50
40 PRINT A:GOTO 20
50 PRINT"QUIT NOW (Y/N)? ";
60 CURSOR ON:KEY WAIT A$:CURSOR OFF
70 IF A$="Y" THEN PRINTA$:STOP KEY ON:END
80 IF A$="N" THEN PRINTA$:GOTO 20 ELSE 60
```

SWAP

PURPOSE:

Exchange the values of two variables of the same data type.

SYNTAX:

SWAP *variable1, variable2*

DESCRIPTION:

The SWAP statement efficiently exchanges the values of two variables. This eliminates the need of a third variable to make the swap. String variables are exchanged by simply swapping the string pointers of both variables. SWAP is very useful in algorithms that sort array data.

variable1 is the first variable name that will be exchanged.

variable2 is the second variable name that will be exchanged.

variable1 & *variable2* must be of equal type, such as floating point to floating point, integer to integer, string to string. If the variables are of different types, a TYPE MISMATCH ERROR will occur.

EXAMPLE:

```
SWAP A,B          :'A & B EXCHANGE NUMERIC VALUES
SWAP A$(1),A$(2)  :'A$(1) & A$(2) EXCHANGE STRING VALUES
SWAP X%,Y%        :'X% & Y% EXCHANGE NUMERIC VALUES
SWAP N$,K$        :'N$ & K$ EXCHANGE STRING VALUES
```

SYS

PURPOSE:

Call a machine language subroutine (augmented).

SYNTAX:

SYS *address [,a [,x [,y [,p]]]]*

DESCRIPTION:

SYS is a CBM BASIC statement that has been augmented to optionally include values for processor registers A, X, Y and processor status flags.

a, x and *y* (0-255) are the processor registers that store one byte each.

p (0-255) is the one byte binary value that contains the processor status flags as defined below. Flags can be combined by adding each bit value.

```
Bit 7 (128) = Negative
Bit 6 (64)  = Overflow
Bit 5 (32)  = Not Used
Bit 4 (16)  = BREAK
Bit 3 (8)   = Decimal
Bit 2 (4)   = Interrupt Disable
Bit 1 (2)   = Zero
Bit 0 (1)   = Carry
```

If you want to clear any flag before the call it is safe set the corresponding bit value to zero or set them all to zero using the byte value of 0. However, be careful when setting the flags to 1 to avoid disabling interrupts, causing a break or entering decimal mode. This can cause unwanted effects on your system.

When the call returns the state of these registers can be read by PEEKing the value as follows: A=PEEK($30C), X=PEEK($30D), Y=PEEK($30E), P=PEEK($30F).

EXAMPLE:

```
SYS $E9FF,0,INF(2) :'CALL KERNAL TO CLEAR CURRENT LINE
SYS $FFD2,ASC("A") :'CALL KERNAL TO OUTPUT THE LETTER "A" TO CURRENT DEVICE
SYS 64738          :'CALL KERNAL TO RESET COMPUTER
SYS 65520,0,10,5,0 :'CALL KERNAL TO POSITION CURSOR AT COL 10, ROW 5, (CARRY CLR)
'CALL KERNAL TO READ CURSOR LOCATION (CARRY SET) AND GET ROW AND COL
SYS 65520,0,0,0,1:X=PEEK($30D):Y=PEEK($30E)
SYS 49152,0,0,0    :'CALL USER ML ROUTINE WITH A,X,Y REGS LOADED WITH ZERO
```

TIME and **TIME$**

PURPOSE:

To get or set the Time of Day (TOD).

SYNTAX:

TIME CLR
T = **TIME**
T$ = **TIME$**
**TIME$** = *tod$*

DESCRIPTION:

TIME returns the number of seconds since midnight. TIME$ returns the string representation of the clock (TOD #2) in 24-hour format (military time). It also is used to set the clock to a specified time. When the computer is switched on the time starts at 01:00:00 (1 AM). When the clock reaches midnight it rolls over to 12 AM (00:00:00). This clock is based on a hardware real time clock which continues keep time even after a soft reset of the computer. It is not driven or affected by software interrupts and thus keeps time more accurately than the CBM BASIC's jiffy clock referenced by variables TI and TI$.

TIME CLR can be used to set the clock to midnight 00:00:00 (12 AM).

T is a floating point variable to capture the current number of seconds since midnight from the TOD clock. It is accurate to 1/10th of a second. The minimum value possibly returned by TIME is 0.0 and the maximum is 86399.9.

T$ is a string variable to capture the current time string from the TOD clock.

tod$ is an 8 character string in the format "hh:mm:ss" used to set the TOD clock. The time is represented as a 24-hour clock (00:00:00 to 23:59:59). *hh* is the hours (00-23), *mm* is the minutes (00-59) and *ss* is the seconds (00-59). If the string supplied is not in the exact format a TYPE MISMATCH ERROR will occur. The clock immediately advances forward from the time that was set.

EXAMPLE:

```
T = TIME            :'GET THE NUMBER OF SECONDS SINCE MIDNIGHT IN VAR T
PRINT TIME$         :'PRINT THE CURRENT TIME IN STRING FORMAT
TIME$ = "13:00:00"  :'SET THE TIME TO 1 O'CLOCK PM
TIME CLR            :'SET THE TIME TO MIDNIGHT
```

TEXT

PURPOSE:

To print characters on a bitmap screen or to set display mode to normal text and color mode.

SYNTAX:

TEXT
TEXT *x, y, string$ [,charset] [,sx] [,sy] [,plottype] [,color]*

DESCRIPTION:

TEXT is a dual purpose statement. When used with no parameters the SCREEN is turned ON (visible) and set to page 0 for standard text, font and color mode. TEXT with parameters will print characters anywhere on a bitmap screen with varying sizes and character sets.

x (0-319 hires, 0-159 multicolor), y (0-199) are the bitmap coordinates where the top left-hand corner of the first character will print. The last plotted point can be referenced with x=INF(18) and y=INF(19).

string$ is the string of characters to be printed on the bitmap screen. There are 29 ASCII control characters supported. Insert is not supported. Any string that does not fit on the screen will be truncated (no wrapping).

charset (0-3) is the character set to use when printing the letters or symbols.

CHARSET	DESCRIPTION
0	Upper-case and symbols (default)
1	Reverse of set 0
2	Lower-case and symbols
3	Reverse of set 2

sx, sy (0-31, default 1) select the size of the x and y dimensions. The numbers here are multiples of the normal size. In multicolor mode the text clarity will be reduced and can be corrected by setting *sx* = 2.

plottype (0-3) is how the characters will be printed (see PLOT statement).

color (0-15 hires, 1-3 multicolor) is the text color (See MAPCOL statement).

EXAMPLE:

```
TEXT                          :'RETURN TO NORMAL TEXT, FONT AND COLOR MODE
TEXT 0,0,"CBM",0,1,5          :'UPPER CHARS AT TOP LEFT CORNER WITH TALL LETTERS
TEXT INF(18),INF(19)," BASIC",1 :'RVS CHARS AT CURRENT LOCATION DEFAULT SIZE
```

TRACE

PURPOSE:

Trace the execution of program statements.

SYNTAX:

TRACE
TRACE *[lineNum]*

DESCRIPTION:

This command is used for debugging purposes. When a program is being traced, the line that is being executed is displayed at the top of the screen. Pressing the shift key will execute the first statement on the line. Each press of the shift key will execute the next statement on the line before advancing to the next line. If a control flow statement (GOTO, GOSUB, RETURN, RESUME) is encountered then the target line will be displayed after it executes.

lineNum (optional) is used to start the program at a specific line number.

TRACE (like RUN) clears all variable data before executing the program.

Anytime the program goes back into direct mode the TRACE is disabled so there is no need to turn it off. This happens when one of the following occurs:
1. Program ends
2. STOP statement encountered in program
3. Run/Stop key pressed
4. An error occurs without error trapping enabled (see ON ERR statement)

NOTE: TRACE will consume the first two lines on the screen. You may have to adjust your program to account for this. Also, if the line listed exceeds 80 characters then the third line fragment will remain on the screen. It is preferred to not exceed 80 characters of BASIC text per line.

EXAMPLE:

```
TRACE            :'RUN PROGRAM AND TRACE THE LINES
TRACE 100        :'START TRACING AT LINE 100
```

VALB(), VALH(), VALO()

PURPOSE:

To provide decimal conversion of string values in base 2, 8 or 16.

SYNTAX:

B = VALB(*binary$***)**
H = VALH(*hexadecimal$***)**
O = VALO(octal*$***)**

DESCRIPTION:

These functions are similar to VAL function for converting numbers stored in a string to its numeric decimal value. Zero is returned if conversion fails.

The string parameter (*binary$, hexadecimal$, octal$*) represents the value with a specific base to be converted to a 32-bit decimal number. The conversion stops at the end of the string or the first invalid character for the base. All spaces are ignored. Below is the list of string-to-value conversions.

FUNCTION	BASE	BASE NAME	PARAMETER VALUE EXAMPLES*
VALB	2	Binary	"1", "11001", "1100000000000000"
VAL	10	Decimal	"1", "25", "49152"
VALO	8	Octal	"1", "31", "140000"
VALH	16	Hexadecimal	"1", "19", "C000"

*In the table above, the parameter value examples will all result in decimal values 1, 25, 49152 respectively.

To convert decimal to hexadecimal use the HEX$() function, for example:
H$ = HEX$(10)

For literal values in a BASIC program, prefix the value with the associated symbol: % (binary), @ (octal) and $ (hexadecimal). For example:
SYS $FFD2,$41 : PRINT %101010100110 : O = @77

NOTE: MDBASIC does not provide a function to convert a decimal value to a binary or octal string.

EXAMPLE:

```
PRINT VALH("FCE2"), VALB("1000"), VALO("30")
 64738    8        24
```

VARS

PURPOSE:

List the current variable names and values that have been assigned.

SYNTAX:

VARS

DESCRIPTION:

VARS is short for variables. It is a useful tool when debugging a program. Program execution can be halted either by END, STOP or the BREAK key, and VARS will display all the current variable names and their values in the order they were dimensioned.

VARS does not list any array variables, being that there are usually far too many values assigned in this type of variable storage. Use an inline FOR/NEXT loop to print array variables if needed.

NOTE: Using the shift key pauses the listing allowing the user time to scan through the list for the desired variables. Also, you can DUMP the variables to a printer by using DUMP VARS.

EXAMPLE:

```
VARS

MB$="MARK BOWREN"
X=160
SP$="               "
A=9693868
N%=25
NU$=""

READY.
```

VOICE

PURPOSE:

To set a frequency (pitch) for any voice or to clear the SID registers.

SYNTAX:

VOICE *voice, frequency*
VOICE CLR

DESCRIPTION:

VOICE is a dual function statement that allows initialization of the SID chip and setting the frequency (pitch) of a voice. The SID chip provides three separate oscillator channels called voices. Each voice has 16-bit frequency resolution, waveform control, envelope shaping, oscillator synchronization, and ring modulation. In addition, each voice can be optionally routed to a programmable filter to further enhance the sound (See the FILTER statement).

There are 5 basic steps to produce sounds:
1. Set the volume using the VOL statement.
2. Select the frequency output using the VOICE statement.
3. Set the envelope parameters using the ENVELOPE statement.
4. Select the type of waveform and start the desired part of the envelope cycle using the WAVE statement.

voice (1-3) selects which voice will have the frequency.

frequency (0.0 - 3994.937 {NTSC-M} or 3848.598 {PAL-B}) expressed in Hertz (Hz), selects what the frequency output of the voice. Fractional values can be specified but the accuracy of the SID chip is limited. This value can be changed at any time to achieve special sound effects.

VOICE CLR clears the registers that are used to control the sounds for all three voices. This is referred to as SID initialization. When executed, all sounds will be turned completely off, and all 24 SID sound registers are set to 0.

EXAMPLE:

```
10 VOICE CLR          :'INITIALIZE SID CHIP
20 VOICE 1,2000       :'VOICE 1 AT 2K HZ
30 ENVELOPE1,0,0,15,0 :'SUSTAIN VOLUME TO MAX
40 VOL 15             :'SET VOLUME TO MAX
50 WAVE 1,1,1         :'START SOUND WITH TRIANGLE WAVEFORM
```

VOL

PURPOSE:

Set the volume of the Sound Interface Device (SID) for all voices.

SYNTAX:

VOL *volume*

DESCRIPTION:

VOL is used to turn up or down the volume of all three voices simultaneously. Use the ENVELOPE statement to select separate sustain volumes for each voice.

volume (0-15) sets the volume. A value of 0 is off and 15 is loudest.

VOICE CLR will set all SID registers to 0 which results in a volume of 0. No sound will be heard unless the volume to an audible level.

A voice will produce little or no sound if any one of the following is true:
1. Voice frequency (pitch) too low or 0 (VOICE statement)
2. Voice waveform is disabled (WAVE statement)
3. Voice was not gated to start (WAVE statement)
4. Voice finished the decay cycle (ENVELOPE statement)
5. Voice sustain volume too low or 0 (ENVELOPE statement)
6. Voice is filtered using the No-pass filter (FILTER statement)
7. Volume too low or 0 (VOL statement)
8. Cable connection or TV/monitor volume too low or off
9. Faulty SID chip or output transistor

EXAMPLE:

```
0 '** SAMPLE SOUND PROGRAM OF RANDOM STATIC **
10 VOICE CLR           :'CLEAR ALL SID REGISTERS
20 ENVELOPE 1,0,0,15,0 :'MAX SUSTAIN VOLUME
30 VOICE 1,2000        :'SET VOICE 1 AT 2KHZ
40 WAVE 1,8,1          :'NOISE WAVEFORM; START ADS CYCLE
50 VOL 15              :'VOLUME AT MAX
60 WAIT RND(TI)*25     :'RANDOM DELAY LENGTH
70 VOL 0               :'TURN OFF NOISE ABRUPTLY
80 WAIT RND(TI)*25     :'RANDOM WAIT LENGTH
90 GOTO 50             :'LOOP
```

WAIT

PURPOSE:

To perform a timed pause in program execution or wait for a value in a specified memory location.

SYNTAX:

WAIT *location, mask1 [,mask2]*
WAIT *jiffies*

DESCRIPTION:

WAIT is a CBM BASIC statement that has been augmented to also perform a timed pause in program execution which can be aborted by pressing the run-stop key. The original CBM BASIC WAIT statement causes program execution to suspend until a given memory address matches a specified bit pattern and can only be aborted by pressing the run-stop & restore key combination.

location (0-65535) is the memory address to compare the mask values.

mask1 (0-255) is the value to be ANDed with the memory location’s value.

mask2 (0-255) is an optional value to exclusive-OR with the result of *mask1*.

mask1 "filters-out" any bits that you don't want to test. Where the bit is 0 in *mask1*, the corresponding bit in the result will always be 0. The *mask2* value flips any bits, so that you can test for an off condition as well as an on condition . Any bits being tested for a 0 should have a 1 in the corresponding position in *mask2*.

jiffies (0-65535) is an unsigned integer of the number jiffies (approximately one sixtieth of a second) to wait. Any value outside this range will result in an ILLEGAL QUANTITY ERROR.

There are many reasons for using a timed delay including waiting for the user to read the screen, a musical note to complete or slow down program execution.

EXAMPLE:

```
WAIT 300           :'WAIT FOR 5 SECONDS (300 JIFFIES)
WAIT $DD01,$80     :'WAIT TILL $DD01 IS EXACTLY $80
WAIT $DD01,$80,$7F :'WAIT TILL BIT 7 IS SET IN MEM $DD01
```

WAVE

PURPOSE:

To select a waveform for a SID voice and start an envelope cycle.

SYNTAX:

WAVE *voice, waveform [,gate [,sync [,ring [,disable]]]]*

DESCRIPTION:

WAVE is primarily used to select a waveform for a voice and start an envelope cycle. It also provides oscillator synchronization and ring modulation. It can also completely disable a voice oscillator. The optional parameters (*gate, sync, ring, disable*) are cumulatively required and default to 0 when omitted.

voice (1-3) is the voice to apply the waveform settings.

waveform (0-8) selects the waveform for the voice. Each waveform contains varying harmonic content that affects the tone quality of the sound. The waveforms available are as follows:

VALUE	WAVEFORM
0	none
1	triangle
2	saw tooth
3	saw tooth + triangle
4	pulse
5	pulse + triangle
6	pulse + saw tooth
7	pulse + saw tooth + triangle
8	noise

Refer to the PULSE statement for setting the duty cycle of the pulse waveform.

gate (1 or 0, default 0) is a Boolean expression used to start one of the two envelope cycles. The first part of the envelope is the Attack, Decay, Sustain cycle. When *gate is set to* 1, the output of the selected voice will follow its envelope settings. After rising to a peak and declining to the sustain volume, the volume will continue at the sustain level until *gate* is set to 0, which starts the release cycle. Thus, *gate* has two functions:
1 = start the attack/decay/sustain cycle
0 = start the release cycle

Refer to the ENVELOPE statement for to configure a voice envelope.

sync (1 or 0, default 0) is a Boolean expression used to synchronize the fundamental frequency of the specified *voice* with its associated sync voice, allowing the creation of complex harmonic structures from the selected *voice*. When enabled, the synchronized voice's frequency affects the output of the selected voice. The following is the synchronization combinations:

a. Voice 1 syncs with voice 3
b. Voice 2 syncs with voice 1
c. Voice 3 syncs with voice 2

ring (1 or 0, default 0) is a Boolean expression used to replace the triangle waveform with a ring modulated combination of the specified *voice* with its associated ring *voice*. This produces non-harmonic overtone structures that are useful for creating bell or gong sound effects. The *voice* selected can ring with one other voice as follows:

a. Voice 1 rings with voice 3
b. Voice 2 rings with voice 1
c. Voice 3 rings with voice 2

disable (1=disable, 0=enable, default 0) is a Boolean expression used to disable the oscillator of the selected voice. This can be useful when generating very complex waveforms like those used for speech synthesis.

EXAMPLE:

```
WAVE 1,1           :'VOICE 1 HAS A TRIANGLE WAVEFORM
WAVE 3,0,0,0,0,1   :'DISABLE VOICE 3 OSCILLATOR
WAVE 3,2,1         :'VOICE 3 HAS A SAW TOOTH WAVEFORM; START ADS CYCLE


0 'SIREN SOUND
10 VOICECLR:VOL15
20 VOICE1,900
25 ENVELOPE1,5,0,15,5
30 WAVE1,1,1
40 FORI=800TO1000:VOICE1,I:NEXT
50 WAIT10
60 FORI=1000TO800STEP-1:VOICE1,I:NEXT
70 C=C+1:IFC<2THEN40
75 WAVE1,1,0:WAIT20
80 VOICECLR
```

APPENDIX A

LOADING/SAVING SCREENS AND FONTS

SAVE"filename", *device*, *secondary*

This is the CBM BASIC standard syntax for saving/loading programs. Valid secondary address numbers are 0-31 for serial devices; 32-127 for other devices.

MDBASIC utilizes the secondary address numbers 16, 17 and 18. These numbers define which type of information to save or load. The secondary addresses and the information they represent are as follows:

```
16 = Video Matrix    - screen text and colors (2K)
17 = Character Set   - custom character set (4K)
18 = Bitmap Graphics - plotted dots and colors (10K)
```

A file saved with these secondary addresses can only be loaded by the same secondary address. The following are some examples of loading & saving.

```
SAVE"SCREEN",8,16    :'save current text screen with colors
LOAD"FONT",8,17      :'load redefined text characters, use SCREEN 1 to show
LOAD"BITMAP",8,18    :'load and show a bitmap screen (hires or multicolor)
```

NOTE: If a text screen in redefined character mode is saved, only the scan codes (poke codes) on the screen are saved. The character shapes must be saved in a separate file using secondary address 17 with a different filename.

When loading/saving a text screen (secondary=16) the data is loaded loaded/saved from the RAM of the current screen page. Below is the page memory map:

PAGE	SCREEN RAM
0	$0400-$07E7
1	$C000-$C3E7
2	$C400-$C7E7
3	$C800-$CBE7
4	$CC00-$CFE7

BINARY SAVE

You can specify the memory locations for a save operation as follows:

SAVE *start, end, filename$, device, secondary*

NOTE: the the filename$, device and secondary parameters are optional. The default device is 1 (tape) as usual.

```
SAVE $C000, $CFFF           :'SAVE HIRAM TO TAPE – NO FILENAME
SAVE 49152, 53247,"HIRAM",8 :'SAVE HIRAM TO DISK – FILENAME REQUIRED
```

APPENDIX B

ERROR CODES & MESSAGES

Error numbers 1-30 are the CBM BASIC errors. Error numbers 31-35 are MDBASIC errors while 0 and 36-127 are user-defined..

To manually cause an error use the ERR statement followed by the number. Any attempt to raise an error outside the valid range will always result in error 14 (Illegal Quantity)

ERROR#	MESSAGE
0	USER DEFINED
1	TOO MANY FILES
2	FILE OPEN
3	FILE NOT OPEN
4	FILE NOT FOUND
5	DEVICE NOT PRESENT
6	NOT INPUT FILE
7	NOT OUTPUT FILE
8	MISSING FILE NAME
9	ILLEGAL DEVICE NUMBER
10	NEXT WITHOUT FOR
11	SYNTAX
12	RETURN WITHOUT GOSUB
13	OUT OF DATA
14	ILLEGAL QUANITY
15	OVERFLOW
16	OUT OF MEMORY
17	UNDEF'D STATEMENT
18	BAD SUBSCRIPT
19	REDIM'D ARRAY
20	DIVISION BY ZERO
21	ILLEGAL DIRECT
22	TYPE MISMATCH
23	STRING TOO LONG
24	FILE DATA
25	FORMULA TOO COMPLEX
26	CAN'T CONTINUE
27	UNDEF'D FUNCTION
28	VERIFY
29	LOAD
30	BREAK
31	ILLEGAL SPRITE NUMBER
32	MISSING OPERAND
33	ILLEGAL VOICE NUMBER
34	ILLEGAL COORDINATE
35	CAN'T RESUME
36-127	USER DEFINED

APPENDIX C

ASCII TABLE OF CHARACTERS

The below table is a partial list of ASCII and screen codes if applicable. ASCII codes are printed with the CHR$() function, screen codes are displayed with POKE.

ASCII	HEX	BINARY	PRINTS	SCREEN CODE
0	00	00000000		
1	01	00000001		
2	02	00000010		
3	03	00000011		
4	04	00000100		
5	05	00000101	WHITE	
6	06	00000110		
7	07	00000111		
8	08	00001000	DISABLE SHIFT/LOGO	
9	09	00001001	ENABLE SHIFT/LOGO	
10	0A	00001010		
11	0B	00001011		
12	0C	00001100		
13	0D	00001101	RETURN	
14	0E	00001110	SWITCH TO LOWER CASE	
15	0F	00001111		
16	10	00010000		
17	11	00010001	CURSOR DOWN	
18	12	00010010	RVS ON	
19	13	00010011	CURSOR HOME	
20	14	00010100	DELETE	
21	15	00010101		
22	16	00010110		
23	17	00010111		
24	18	00011000		
25	19	00011001		
26	1A	00011010		
27	1B	00011011		
28	1C	00011100	RED	

ASCII	HEX	BINARY	PRINTS	SCREEN CODE
29	1D	00011101	CURSOR RIGHT	
30	1E	00011110	GREEN	
31	1F	00011111	BLUE	
32	20	00100000	SPACE	32
33	21	00100001	!	33
34	22	00100010	"	34
35	23	00100011	#	35
36	24	00100100	$	36
37	25	00100101	%	37
38	26	00100110	&	38
39	27	00100111	'	39
40	28	00101000	(	40
41	29	00101001	)	41
42	2A	00101010	*	42
43	2B	00101011	+	43
44	2C	00101100	,	44
45	2D	00101101	-	45
46	2E	00101110	.	46
47	2F	00101111	/	47
48	30	00110000	0	48
49	31	00110001	1	49
50	32	00110010	2	50
51	33	00110011	3	51
52	34	00110100	4	52
53	35	00110101	5	53
54	36	00110110	6	54
55	37	00110111	7	55
56	38	00111000	8	56
57	39	00111001	9	57
58	3A	00111010	:	58
59	3B	00111011	;	59
60	3C	00111100	<	60
61	3D	00111101	=	61
62	3E	00111110	>	62
63	3F	00111111	?	63

ASCII	HEX	BINARY	PRINTS	SCREEN CODE
64	40	01000000	@	0
65	41	01000001	A	1
66	42	01000010	B	2
67	43	01000011	C	3
68	44	01000100	D	4
69	45	01000101	E	5
70	46	01000110	F	6
71	47	01000111	G	7
72	48	01001000	H	8
73	49	01001001	I	9
74	4A	01001010	J	10
75	4B	01001011	K	11
76	4C	01001100	L	12
77	4D	01001101	M	13
78	4E	01001110	N	14
79	4F	01001111	O	15
80	50	01010000	P	16
81	51	01010001	Q	17
82	52	01010010	R	18
83	53	01010011	S	19
84	54	01010100	T	20
85	55	01010101	U	21
86	56	01010110	V	22
87	57	01010111	W	23
88	58	01011000	X	24
89	59	01011001	Y	25
90	5A	01011010	Z	26
91	5B	01011011	[	27
92	5C	01011100	£	28
93	5D	01011101	]	29
94	5E	01011110	↑	30
95	5F	01011111	←	31

Screen codes from 128-255 are reversed images of codes 0-127.

See the Commodore 64 Programmer’s Reference Guide, Appendix C for a complete list.

APPENDIX D

RS-232 STATUS CODES

The status of RS-232 data port can be determined by using the CBM BASIC reserved variable ST. This variable will be set to 0 after it is read so you if need to use it more than once then store the result in another variable and use that variable for analysis. The status value is described below:

```
Bit 7: 1 = (128) Break Detected
Bit 6: 1 =  (64) DTR (Data Set Ready) Signal Missing
Bit 5: Unused
Bit 4: 1 =  (16) CTS (Clear to Send) Signal Missing
Bit 3: 1 =   (8) Receiver Buffer Empty
Bit 2: 1 =   (4) Receiver Buffer Overrun
Bit 1: 1 =   (2) Framing Error
Bit 0: 1 =   (1) Parity Error
```

The user is responsible for checking the status and taking appropriate action. If, for example, you find that Bit 0 or 1 is set when you are sending, indicating a framing or parity error, you should resend the last byte. If Bit 2 is set, the SERIAL READ statement is not being executed fast enough to empty the buffer. MDBASIC should be able to keep up at 1200 baud safely, 2400 max. If Bit 7 is set, you will want to stop sending, and execute SERIAL READ to see what is being sent.

The limitations of the communication speed on the serial port is due to the implementation of the buffered read/write process that is driven by an IRQ handler.

APPENDIX E

GLOSSARY

acronym

A word formed by the initial letters of words or by initial letters plus parts of several words.

address

A name, label, or number identifying a register, location or unit where information is stored.

algebraic language

A language whose statements are structured to resemble the structure of algebraic expression. Fortran is an algebraic language.

algorithm

A set of well-defined rules or procedures to be followed in order to obtain the solution of a problem in a finite number of steps. An algorithm can involve arithmetic, algebraic, logical and other types of procedures and instructions. All algorithms must produce a solution within a finite number of steps.

alphanumeric

A contraction of the words alphabetic and numeric; a set of characters including letters, numerals, and special symbols.

argument

1. A type of variable whose value is not a direct function of another variable. It can represent the location of a number in a mathematical operation, or the number with which a function works to produce its results.
2. A known reference factor that is required to find a desired item (function) in a table. For example, in the SQR(x) function, x is the argument, which determines the square root value returned by this function.

array

1. An organized collection of data in which the argument is positioned before the function.
2. A group of items or elements in which the position of each item or element is significant. A multiplication table is a good example of an array.

APPENDIX E (continued)

ASCII

Acronym for American Standard Code for Information Interchange. ASCII is a standardized 8-bit code used by most computers for interfacing.

assembler

The computer program that produces a machine language program which may then be directly executed by the computer's microprocessor.

assembly language

A symbolic language that is machine-oriented rather than problem oriented. An assembly language program is converted to machine code by an assembler.

BASIC

Acronym for Beginner's All-Purpose Symbolic Instruction Code. BASIC is a computer programming language developed at Dartmouth College as an instructional tool in teaching the fundamental programming concepts.

baud

A unit of measurement of data processing speed. The speed in bauds is the number of signal elements per second. Typical baud rates are 50, 75, 110, 300, 1200, 2400. The C64 does not support higher rates.

big-endian

See definition of **endian**.

binary

1. A characteristic or property involving a choice or condition in which there are two possibilities.
2. A numbering system which uses 2 as its base instead of 10 as in the decimal numbering system. This system uses only two digits, 0 and 1.
3. A device whose design uses only two possible states or levels to perform its functions. A computer executes programs in binary form.

bit

A contraction of "binary digit" which is expressed in the binary digits of 0 and 1, and is the smallest unit of measurement recognizable by the computer. There are eight bits in a byte.

Boolean logic

Developed by the British mathematician George Boole, which has a field of mathematical analysis in which comparisons are made. A programmed instruction can cause a comparison of two fields of data, and modify one of those fields or another field as a result of comparison. Some Boolean operations are OR, AND, NOT.

APPENDIX E (continued)

buffer

A temporary storage area from which data is transfered to or from various devices.

byte

An element of data which is composed of eight data bits plus a parity bit, and represents either one alphabetic or special character, two decimal digits, or eight binary bits.

CPU

The Central Processing Unit (CPU) is the heart of the computer system, where data is manipulated and calculations are performed. The CPU contains a control unit to interpret and execute the program and an arithmetic-logic unit to perform computations and logical processes. It also routes information, controls input, output, and temporarily stores data. A CPU is also known as a microprocessor.

character

Any single letter of the alphabet, numeral, punctuation mark, or other symbols that a computer can read, write, and store. Character is synonymous with the term byte.

compiler

A computer program that translates a program written in a problem-oriented language into a program of instructions similar to, or in the language of the computer.

concatenate

To link together. To join one or more pieces of data to form one piece of data. Files being merged together, or strings of text being added together are two examples.

configuration

In hardware, a group of interrelated devices that constitute a system. In software, the total of the software modules and their inter-relationships.

constant

A never changing value or data item.

debug

The process of checking the logic of a computer program to isolate and remove mistakes from the program or other software.

APPENDIX E (continued)

default

An action or value that the computer automatically assumes, unless a different instruction or value is given.

delimiter

A character that marks the beginning or end of a unit of data on a storage medium. Commas, semicolons, periods, spaces, and quotations are used as delimiters to separate and organize items of data.

DOS

A Disk Operating System (DOS) is a collection of procedures and techniques that enable the computer to operate using a disk drive system for data entry and storage.

endian

Endian refers to the order in which bytes within a word of binary data appear in consecutive computer memory addresses. The numeric significance of the byte having the lowest address determines its endianness. When the first byte is more significant than the second byte it is called big-endian. When the second byte is more significant than the first byte it is called little-endian.

fixed disk

A hard disk enclosed in a permanently-sealed housing that protects it from environmental interference. It is like a floppy disk except it can store and retrieve a larger amount of data at a faster speed.

floating point

A method of calculation in which the computer or program automatically records, and accounts for, the location of the radix point (numbers to the right of the decimal point). The programmer need not consider the radix location.

garbage collection

A process that the computer takes when too many strings have been assigned, and then reassigned, leaving massive amounts of useless data. When the process takes place, the computer will seemed to have "locked up". There is no indication to the user what is going on and it is a long process.

hard copy

Computer data that is printed to paper.

hardware

The physical equipment that comprises a system.

APPENDIX E (continued)

hexadecimal

A number system with a base of 16. The numbering system ranges from 0-9 and then A-F, rather that 0-9 only on the decimal system.

interpreter

A program that reads, translates and executes a user's program, such as one written in the BASIC language, on line ate a time. A compiler on the other hand, reads and translates the entire user's program before executing.

integer

A complete entity, having no fractional part. The whole or natural number. For example, 65 is an integer, 65.1 is not.

K

The symbol signifying the quantity of 2 to the 10th power=1024. K is sometimes confused the symbol K (kilo) which is equal to 1000.

little-endian

See definition of **endian**.

logarithm

A logarithm of a given number is the value of the exponent indicating the power required to raise a specified constant, known as the base, to produce that given number. That is, if B is the base, N is the given number and L is the logarithm, then BL=N.

M

The symbol signifying the quantity 1,000,000 (10~6). When used to denote storage, it more precisely refers to 1,048,576 (2~20).

mantissa

The fractional or decimal part of a logarithm of a number. For example, the logarithm of 163 is 2.212. The mantissa is 0.212, and the characteristic is 2.0.

nybble

Half of a byte which is 4 bits so there are two nybbles in every byte.

null

In the context of traditional BASIC (not an object-oriented language) null usually refers to a blank or zero value assigned to a variable. For example, a string of zero length or the ASCII value of 0.

APPENDIX E (continued)

NTSC *(NATIONAL TELEVISION STANDARD CODE)*

The U.S. standard for timing of the raster scan lines. NTSC has 262 horizontal lines which make a display. Only 200 of these lines are visible (50-249) on the Commodore 64. The European (PAL) standard has 312 lines, making the clock speed slower to sync the sixtieth of a second interrupt.

octal

A representation of values or quantities with octal numbers. The octal number system uses eight digits: 0-7, with each position in an octal numeral system representing a power of 8. The octal system is used in computing as simple means of expressing binary quantities.

operand

In context of a BASIC statement, an operand is a parameter for the statement that resolves to a value. The data type and value must follow the rules of the statement to which it belongs.

operating system

An organized group of computer instructions that manage the overall operation of the computer.

parameter

A variable that is given a value for a specific program to process. A definable item, device, or system.

PAL *(Phase Alternating Line)*

European standard for television scan lines (See NTSC).

parity

In RS-232, parity is a method of detecting errors in a stream of bits.

peripheral

An external input/output, or storage device.

pixel

The acronym for picture element. A pixel is a single dot on a monitor that can be addressed by a single bit.

RAM *(Random Access Memory)*

The system's high speed work area that provides access to memory storage locations by using a system of vertical and horizontal coordinates. The computer can write or read information to RAM faster than any other input/output process.

APPENDIX E (continued)

raster

On a graphics display screen a raster unit is the horizontal or vertical distance between two adjacent addressable points on the screen. There are 262 horizontal lines which make up the American (NTSC) standard display screen (312 lines in European or PAL standard screens). Every one of these lines is scanned and updated 60 times per second. Only 200 of these lines (50-249) are part of the visible display.

ROM *(Read Only Memory)*

A type of memory that contains permanent data or instructions. The computer can read but not write to ROM.

real number

An ordinary number, either rational or irrational; a number in which there is no imaginary part, a number generated from the single unit, 1; any point in a continuum of natural numbers filled in with all rationales and all irrationals and extended indefinitely, both positive & negative.

SID

The Sound Interface Device (SID) is a custom music synthesizer and sound effects generator chip in the C64. It provides three separate music channels (voices) each with a 16-bit frequency resolution, waveform control, envelope shaping, oscillator synchronization, and ring modulation. In addition, programmable high-pass, low-pass, and band-pass filters can be set and enabled or disabled for each sound channel.

software

A list of instructions that, when executed direct the computer to perform certain tasks.

sprite

A graphical Movable Object Block (MOB) that is independent of the text or graphics display. The C64’s VIC-II chip supports eight sprites with the following features:

- 24 horizontal dot by 21 vertical dot size
- Individual color control for each sprite
- Multicolor mode
- 2X axis magnification in horizontal, vertical, or both
- Selectable sprite to background priority
- Fixed sprite-to-sprite priorities
- Sprite-to-sprite collision detection
- Sprite-to-text collision detection

syntax

Rules of statement structure in a programming language.

APPENDIX E (continued)

toggle

Alternation of function between two stable states.

token

Each keyword entered into a BASIC program is converted to a binary storage format which uses less space than plain text and executes faster. This binary value is a single-byte that represents each keyword.

truncation

To end a computation according to a specified rule; for example, to drop numbers at the end of a line instead of rounding them off, or to drop operands off a statement syntax when variables do not pertain to the task that is to be performed.

variable

In the context of a BASIC program a variable is a named value that can be changed. Variables are defined by the type of data they hold (string, integer, float or array). The C64 is limited to only two characters per name and will ignore additional characters if present. Variable names must begin with an letter and the second (optional) character can be a letter or numeric digit.

video matrix

An array (40 x 25) of memory locations used to display either text or color for bitmap graphics. In text mode each byte contains a value called a screen code which is an offset index to the binary pattern stored in character ROM or user RAM to display an 8 x 8 character. In bitmap mode each byte used to select color. Both modes are affected my multicolor mode.

APPENDIX F

SAMPLE PROGRAMS

```
10 '*** EXTENDED BASIC DEMO 1 ***
20 '
30 '  **** BY: MARK BOWREN ****
40 '
50 SP=100
55 VOICECLR:COLOR14,0,0:SCREENCLR
60 VOL15:ENVELOPE1,0,9,0,0:VOICE1,1000
65 SPRCOL 2,6
70 SPRITE0,1,5,1,13,0,1
80 FORI=832TO832+63:READA:POKEI,A:NEXT
90 CURSOR8,3:PRINT"*** BY: MARK BOWREN ***"
110 MOVE0,24,50TO297,50,SP:WAVE1,0:WAVE1,1,1:SP=SP-5
120 MOVE0 TO 297,231,SP:WAVE1,0:WAVE1,1,1:SP=SP-5
130 MOVE 0 TO 24,231,SP:WAVE1,0:WAVE1,1,1:SP=SP-5
140 MOVE 0 TO 24,50,SP:WAVE1,0:WAVE1,1,1:SP=SP-5
150 IF SP >  0 THEN 110
151 CURSOR 16,6:PRINT".FOR .THE:"
152 CURSOR14,15:PRINT".COMMODORE 64.":MOVE0 TO 160,120, 120
155 FORI=0TO21:WAIT30:SPRITE 0 EXPAND (IAND1)*3:NEXT
160 DATA 3,240,0
161 DATA 15,252,0
162 DATA 63,255,0
170 DATA 63,63,0
171 DATA 252,15,0
172 DATA 240,128,255
180 DATA 240,128,252
181 DATA 240,0,240
182 DATA 240,0,192
190 DATA 240,0,0
191 DATA 240,0,64
192 DATA 240,0,80
200 DATA 240,0,84
201 DATA 240,0,85
202 DATA 252,15,0
210 DATA 63,63,0
211 DATA 63,255,0
212 DATA 15,252,0
220 DATA 3,240,0
221 DATA 0,0,0
222 DATA 0,0,0,0
```

Appendix F (continued)

```
0 'DEMO3: MULTICOLOR BITMAP EXAMPLE
1 COLOR14,0,0
10 MAPCOL2,1,6,0
15 SCREENCLR5:SCREEN5,1
20 PLOT 16,10,1,1
21 DRAW"R60,D55,L60,U55"
22 PLOT 16,70,1,2
23 DRAW"R60,D30,L60,U30"
25 PLOT 16,120,1,3
26 DRAW"R60,D30,L60,U30"
30 CIRCLE48,37,14,18,,1,1
32 PAINT48,27,1
35 PAINT17,11,2
45 TEXT 0,0,"JAPAN",0, 2,2, 1,1
46 LINE 0,0 TO 159,199, 1,3
51 MAPCOL 7, 8, 13
52 PLOT150,10,1,3
53 DRAW"R9,D40,L9,U40"
55 LINE 0,199TO159,0, 1,2
91 PLOT 85, 65, 1, 1
92 DRAW"F30,C2,H30,C3,G30,C1,E30"
98 WAIT 200

0 ' *** BITMAP DEMO 4 ***
10 SCREEN CLR 5,1:MAPCOL2,6,15,0
15 TEXT 48,8,"PIE CHART",0,2,2,1,3
20 CIRCLE 79,99,40,60,%00111110,1,3
30 CIRCLE 86,89,40,60,%00110001,1,3
40 PAINT 87,88,1,1
45 TEXT 100,62,"25%",1,1,1,1,3
50 PAINT 78,100,1,2
55 TEXT 55,100,"75%",1,1,1,1,3
60 WAIT 200
65 TEXT

0 ' EXAMPLE USING INF(16)=PLAY INDEX
1 ' ZERO-BASED INDEX OF LAST CHAR
2 ' OF THE NOTE PLAYING/PLAYED
3 ' USEFUL IN BACKGROUND PLAY MODE
5 DIM I,J,A$
9 SCREEN CLR
10 A$="O4W1A60A#BC#D >ABCDEF@"
15 PRINT"PRESS ANY KEY TO STOP":PRINT
20 PRINT" ";A$
25 PLAYA$
30 I=INF(16)
40 CURSOR I:PRINT"^";
41 IFI=LEN(A$)THEN60
45 J=INF(16):IFJ=ITHEN45
50 CURSOR I:PRINT" ";
55 I=J:GETB$:IFLEN(B$)=0THEN30
60 PRINT:PRINT"DONE.":PLAYOFF
```

Appendix F (continued)

```
0 ' DEMO OF SOUND COMMANDS
1 ' ** DIXIE **
2 ' MUSIC BY: JIM BUTTERFIELD
3 DIM I,S,F1,F2,F3
10 SCREEN CLR:CURSOR17,0:COLOR14,0,2
20 PRINT"DIXIE"
30 CURSOR7,2:COLOR,,2:PRINT"MUSIC BY: JIM BUTTERFIELD"
35 SPRITE0,1,1,0,13
40 RESTORE 380
45 FORI=0TO63:READS:POKE832+I,S:NEXT
50 RESTORE 155
55 VOICE CLR:VOL15
60 ENVELOPE1,0,9,0,0
65 ENVELOPE2,2,4,2,4
70 ENVELOPE3,1,2,10,10
75 I=TI
80 WAVE1,1,0 : WAVE3,2,0 : WAVE2,1,0
90 READ S:IF S=0 THEN END
95 READ F1,F2,F3
100 MOVE0,(F1/5)+24, (F2/5)+50
105 VOICE1,F1 : VOICE2,F2 : VOICE3,F3
120 WAVE1,1,1 : WAVE3,2,1 : WAVE2,1,1
135 I=I+(S*10)
140 IF I>TI THEN 140 ELSE 75
150 '** MUSIC DATA **
155 DATA 1, 801, 0, 0
160 DATA 1, 674, 0, 0
165 DATA 2, 535, 337, 133
170 DATA 2, 535, 400, 0
175 DATA 1, 535, 267, 100
180 DATA 1, 600, 0, 0
185 DATA 1, 674, 400, 0
190 DATA 1, 714, 0, 0
195 DATA 2, 801, 337, 133
200 DATA 2, 801, 400, 0
205 DATA 2, 801, 267, 150
210 DATA 2, 674, 400, 168
215 DATA 2, 900, 357, 178
220 DATA 2, 900, 535, 0
225 DATA 2, 900, 450, 133
230 DATA 1, 0, 535, 0
235 DATA 1, 801, 0, 0
240 DATA 2, 900, 357, 178
245 DATA 1, 0, 535, 0
250 DATA 1, 801, 0, 0
255 DATA 1, 900, 450, 168
260 DATA 1, 1010, 0, 0
265 DATA 1, 1070, 535, 150
270 DATA 1, 1201, 0, 0
275 DATA 2, 1348, 337, 133
280 DATA 2, 0, 400, 0
285 DATA 2, 0, 267, 100
290 DATA 1, 1070, 400, 0
```

APPENDIX F (continued)

```
295 DATA 1, 801, 0, 0
300 DATA 2, 1070, 337, 133
305 DATA 2, 0, 400, 0
310 DATA 2, 0, 267, 126
315 DATA 1, 801, 400, 112
320 DATA 1, 674, 0, 0
325 DATA 2, 801, 357, 100
330 DATA 2, 0, 400, 0
335 DATA 2, 0, 300, 150
340 DATA 1, 600, 400, 0
345 DATA 1, 677, 0, 0
350 DATA 2, 535, 337, 133
355 DATA 2, 0, 400, 100
360 DATA 2, 0, 337, 66
365 DATA 2, 0, 0, 0
370 DATA 0
375 '** SPRITE DATA **
380 DATA 224,14,0,248,62,0,255,254,0,255,254,0,199,198,0,192,6,0,248
385 DATA 126,0,223,230,0,192,6,0,192,6,0,192,6,0,192,6,0,240,7,192
390 DATA 248,3,224,112,1,192,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
999 END


0 ' ***RS-232 SERIAL PORT EXAMPLE 1***
5 W=300:S=0:B=13
10 SERIAL OPEN
15 SERIAL WAIT W READ A$ TO B
20 S=ST:IF S<>0 THEN 60
30 PRINT A$;
40 W=50:GOTO 15
60 PRINT"STATUS: ";S
70 SERIAL CLOSE


0 ' ***RS-232 SERIAL PORT EXAMPLE 2***
1 ' 2400 BAUD,8 DTABITS,1 STPBIT,F-DLPX,NO PARITY,3-LN
20 SERIAL OPEN 2400,8,1,0,0,0
30 SERIAL PRINT"THIS IS A TEST OF THE RS-232 WRITE.";TI;CHR$(10)
40 SERIAL CLOSE


0 '***INF FUNCTION EXAMPLE***
10 'PUT TIME AT TOP OF SCREEN & RETURN CURSOR TO ORIGINAL POSITION
20 R=INF(2):C=INF(0) :'REMEMBER CURRENT CURSOR PHYSICAL POSITION
30 CURSOR32,0:PRINT TIME$;:CURSOR R,C
40 IF INF(8) > 0 THEN PRINT"CLOSING ALL FILES":CLOSE FILES
50 SCREEN CLR 5,0:MAPCOL1:PLOT160,100,1
60 X=INF(18):Y=INF(19):P=INF(20) :REM GET LAST PLOT INFO
70 PLOT 160,100,3 :'SET CURRENT COORDINATE WITHOUT PLOTTING A DOT
80 WAIT120:TEXT
90 PRINT"COORDINATE: (";INF(18);",";INF(19);") HAS PIXEL VALUE: ";INF(20)
```

APPENDIX F (continued)

```
0 DIM I,K,K$(4)
1 K$(0)="170 V1   C64"
2 K$(1)="0   V2   C64"
3 K$(2)="3   V3   C64"
4 K$(3)="67  SX   C64SX"
5 K$(4)="100 4064 PET 64/EDUCATOR"
10 K=INF(15)
15 IFK=170THENI=0:ELSEIFK=0THENI=1:ELSEIFK=3THENI=2:ELSEIFK=67THENI=3:ELSEI=4
20 PRINT"STHIS COMPUTER SYSTEM:"
30 PRINT"ID  VER  SYSTEM","VIDEO"
40 PRINT K$(I),
50 IFINF(14)=0THENPRINT"NTSC":ELSEPRINT"PAL"


10 '** SCREEN FINE SCROLLING TEST **
20 DIM I,J,P,PG:PG=1
30 SCREENCLR:PRINT"SWITCHING TO PAGE";PG;:WAIT120
40 IFPG>0THENDESIGNNEW
50 SCREENPG:COLOR14,6,14:SCREENCLR
60 M=INF(10)+39
70 A$="THIS IS A FINE SCROLLING MESSAGE TO DEMONSTRATE HOW IT WORKS..."
80 GOSUB130
90 A$="                                          "
100 GOSUB130
110 SCREEN,8:CURSOR0,0:PRINT"SWITCHING TO PAGE 0";:WAIT120
120 SCREEN0:CURSOR0,0:CURSORCLR:PRINT"DONE":END
130 FOR J=1 TO LEN(A$)
140 P=ASC(MID$(A$,J,1))
150 IFP>63THENP=P-64
160 POKEM,P
170 FOR I=7 TO 0 STEP -1
180 WAIT53265,128:SCREEN,I
190 NEXTI
200 WAIT53265,128
210 SCROLL0,0TO39,0,2,0
220 SCREENPG,7
230 NEXTJ
240 RETURN


0 '***JOYSTICK TESTING***
10 J=1:D=0:DIMA$(10)
20 SCREENCLR
30 A$(1)="UP":A$(2)="DOWN":A$(4)="LEFT":A$(5)="UP & LEFT"
40 A$(6)="DOWN & LEFT":A$(8)="RIGHT":A$(9)="UP & RIGHT":A$(10)="DOWN & RIGHT"
50 PRINT"ENTER JOYSTICK NUMBER (1 OR 2): ";
55 CURSORON:KEYWAITA$:CURSOROFF:PRINTA$
56 IFA$="1"THENJ=1:ELSEIFA$="2"THENJ=2:ELSE55
60 PRINT"WAITING FOR JOYSTICK MOVEMENT"
70 D=JOY(J):IFD=0THEN70
71 F=DAND128:D=DAND15
75 IFF>0THENPRINT"FIRE";:IFD=0THENPRINT:GOTO70:ELSEPRINT" AND ";
80 PRINTA$(D):GOTO70
```

APPENDIX F (continued)

```
0 '***TRUE RANDOM NUMBER GENERATOR***
1 VOICE CLR          :'INIT SID
10 WAVE 3,8          :'NOISE WAVEFORM
15 VOICE 3,3000      :'FREQ 3KHZ
20 X=PEEK(54299)     :'READ RANDOM NUMBER
25 PRINT X           :'DISPLAY NUMBER
30 WAIT 60           :'WAIT 1 SECOND
35 GOTO 20           :'READ ANOTHER RND NUMBER
```

```
0 'RUN 260 FOR ON ERR RESUME NEXT
1 'CHANGE ON ERR GOTO STATEMENT
2 'GOTO 150 FOR RESUME NEXT
3 'GOTO 180 FOR RESUME
4 'GOTO 220 FOR RESUME [LINE#]
10 SCREENCLR
20 D%=0:'DIVISOR TO CAUSE ERROR
30 DIM A(2)
50 ON ERR GOTO 150 :'**HERE**
60 PRINT"START"
70 D%=0:'DIVISOR
80 A(0)=1:A(1)=2/D%:A(2)=3
90 PRINT"ARRAY:";A(0);A(1);A(2)
100 PRINT"END"
110 END
120 GOTO 60
130 STOP
140 '*TEST1***********************
150 GOSUB 340
160 RESUME NEXT:'SKIP
170 '*TEST2***********************
180 GOSUB 340
190 D%=1 :'FIX PROBLEM
200 PRINT "FIX APPLIED"
210 RESUME :'TRY AGAIN
215 '*TEST3***********************
220 GOSUB 340
230 PRINT "ABORT!"
240 RESUME100
250 '*TEST4***********************
260 ON ERR RESUME NEXT
270 A(0)=1:A(1)=2/0 :A(2)=3
280 PRINT"ARRAY:";A(0);A(1);A(2)
290 GOSUB 340
300 ERR CLR
310 GOSUB 330
320 PRINT "END!"
325 END
330 PRINT"ERROR FREE NOW"
340 PRINT"ERR#:";ERR;"LINE#:";ERRL
350 RETURN
```

APPENDIX G

PROGRAMMING GUIDELINES AND SUGGESTIONS:

1. Declare all non-array variables at the top of the program with the most used variables first. This is because the name lookup is a sequential process starting from the first declared variable to the last.

2. Always place array declarations at the end of the declare section. This will avoid moving array data in memory since array data is stored after non-array.

3 Declare constants in a variable if they are heavily used or inside a loop. It is faster to access a variable value than to decode a string of ASCII digits.

4. Integer arrays use less space than non-array integers. If you have many integer variables or constants consider using an array to reference the value instead of individual integer variables.

5. Any subroutine that repeatedly perform string concatenation or reassignment should manually execute a garbage collection by using SYS46374 or the FRE(0) function. This will help avoid an automatic collection which can take a significant amount of time during which the BASIC program is paused.

6. MDBASIC makes use of a few memory locations that are not used by CBM BASIC or the Kernal. Do not use these locations to store any info or sprite data as it will be lost. These include zero-page locations $FB to $FE. Location $0313 is used as bit flags to control which MDBASIC IRQ subroutines are active. Locations $0334 & $0335 are used to store the previous IRQ vector while MDBASIC IRQ driven processes are running. Locations $0336 & $033B temporarily hold vectors to BASIC program text for the normal error handler, custom error handler and key trapping.

USEFUL PEEKS AND POKES

Although MDBASIC was written to help reduce the use of the PEEK and POKE statements, sometimes it is necessary to still use them. The INF function returns most system information to avoid the use of PEEK and all other statements and functions help avoid the use of the POKE statement. Below are useful PEEKs and POKEs for which MDBASIC does not provide a statement or function.

```
POKE 808,234      :'DISABLE STOP/RESTORE KEY COMBO (BREAKS LIST COMMAND)

'WHICH KEYS CAN REPEAT?
POKE 650,128      :'0=CURSOR, INSERT, DELETE & SPACEBAR, 64=NONE, 128=ALL

'SHIFT/LOGO CHARSET SWITCHING ENABLE/DISABLE
POKE 657, 128     :'0=ENABLE, 128=DISABLE
'NOTE: YOU CAN ALSO PRINT CHR$(9) TO ENABLE, CHR$(8) TO DISABLE

C1 = PEEK(53278)  :'SPRITE TO SPRITE COLLISION (8-BITS)
C2 = PEEK(53279)  :'SPRITE TO TEXT COLLISION (8 BITS)
'CLOCK FREQ 50HZ OR 60HZ
IF PEEK(56334) AND 128 THEN F=50 : ELSE F=60
```

BIBLIOGRAPHY

English, Lothar, "The Advanced Machine Language Book for the C64";
© 1984 by Abacus Software, Inc., Grand Rapids, MI

Leemon, Sheldon, "Mapping the Commodore 64";
© 1984 by COMPUTE! Publications, Inc., Greensboro, NC

Mircosoft Corp., "GW-BASIC Users Guide & Technical Reference";
pp. 109-129, © 1988 by Standard Brand Products, USA

Sams, Howard W., "Commodore 64 Programmers Reference Guide";
© 1982 by Commodore Business Machines, Inc., USA