Title page

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 considered an instructional guide for learning programming.

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. All CBM BASIC statements and functions can be used. MDBASIC augments some CBM BASIC statements and adds several additional statements and functions.

MDBASIC is located at memory address $8000 (32768) to $BFFF (43008). Half of the program is written underneath ROM, but there were needs to use 8K of BASIC's RAM area. This leaves the user with 30718 bytes of memory for BASIC programs, but, as you will see when using MDBASIC, your programs will become much shorter and more efficient, which more than makes up for the loss of memory.

MDBASIC comes with 60+ keywords, aiding the programmer with I/O, sound, text, sprites, redefined characters, bitmap screens, and programming-debugging. Many keywords combine with other keywords to provide more capabilities. Certain operands in a statement are optional and can be omitted or skipped using commas. This enables one statement to perform several variations of commands with only one keyword, otherwise there would be over 100 commands to learn.

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 Bowren

INSTALLATION

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

```
LOAD"MDBASIC.PRG",8,1

SYS64738
```

MDBASIC will automatically initialize as if a cartridge is plugged in.

The following is a summary of keywords in MDBASIC categorized by the tasks they perform:

		UTILITIES		
AUTO	DISK	DUMP	DELETE	FILES
FIND	MERGE	OLD	TRACE	VARS
		GRAPHICS		
BITMAP	CIRCLE	DRAW	DESIGN	EXPAND
EXPAND	LINE	MAPCOL	MOVE	MULTI
PAINT	PLOT	SCREEN	SPRITE	TEXT
		AUDIO		
ENVELOPE	FILTER	PLAY	PULSE	VOICE
VOL	WAVE			
		MISCELLANEOUS		
COLOR	CLOSE*	ELSE	ERROR	FILL
KEY	LOCATE	NEW*	POKE*	RESTORE*
RESUME	RETURN*	RUN*	SAVE*	SERIAL
SCROLL	SWAP	SYS*	WAIT*	
		FUNCTIONS		
ERROR	HEX$	INF	INSTR	JOY
KEY	PEN	POT	PTR	ROUND

*Existing CBM BASIC keywords that have been augmented. For more details on the extended capabilities refer to the documentation.

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 Wargames
```

FEATURES & ENHANCEMENTS

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

Device Numbers for loading & saving special data:

2	Text screen & colors including background & border
3	Character designs (after using DESIGN command)
4	Bitmap images complete with color & mode

```
SAVE "MYSCREEN",8,2  :REM SAVE CURRENT TEXT SCREEN TO FILE

LOAD "MYBITMAP",8,4  :REM LOAD & SHOW BITMAP FROM FILE
```

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

```
SAVE $0400,$07f8,"MYSCREEN",8  :REM SAVE 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 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 and OUT OF MEMORY error. It is up to the programmer to be careful about the target memory locations to avoid problems. For example, you would not want to load a file on top of where your program and variables are stored. The safest approach is to load binaries into upper memory ($C000-$CFFF). You can load bitmap graphics directly to $E000-$FFFF (RAM under Kernel ROM) then use the BITMAP statement to show the image.

MDBASIC supports numeric constants of base 2 (binary), 8 (octal) or hexadecimal. The NOT expression short-hand is the exclamation point. REM (remark) short-hand is the apostrophe. See examples below:

```
B = %00001111  :'BINARY 15
H = $FFFF      :'HEX 65535
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. MDBASIC and CBM-BASIC do not enforce names for constant values and must be stored in a variable. It is up to the programmer to know which variables are considered constants. Literals are constants where the value is explicitly entered in the program code and thus cannot change.

Command/Statement 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. This document makes use of brackets [] to denote the parameter is optional. For example:

DISK dos$ [,device#, [string$]]

This implies that the first parameter is a string and is required. The next two parameters are optional. If the second parameter is supplied the third parameter can be omitted. The third parameter requires the first two parameters.
.
When listing the various formats for a statement, the use of the pipe symbol | will be used. The example below means AUTO ON or AUTO OFF is acceptable:

AUTO ON | **OFF**

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

AUTO

PURPOSE:

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

SYNTAX:

AUTO
AUTO ON | OFF
AUTO *[increment]*

DESCRIPTION:

When making a BASIC program, every line must start with a line number. Line numbers may be incremented in any step while programming. AUTO is helpful because it automatically displays the next line number that is to follow the line that was just entered.

AUTO ON turns auto-line numbering on with an increment of 10. This command is not necessary if an increment other than 10 is desired.

AUTO OFF turns auto-line numbering off.

increment is the optional (default 10) number of lines that will be added to the last line entered to calculate the next line number. Line numbers range from 0-63999, so it is not advisable to have increments higher than 1000, especially with long programs.

The programmer must type in the first line number, or hit return on the current working line number for the next line to be displayed.

EXAMPLE:

```
AUTO 10     :REM sets auto-line numbering with increments of 10
AUTO 1000   :REM sets auto-line numbering with increments of 1000
AUTO OFF    :REM disables auto-numbering
```

BITMAP

PURPOSE:

Selects bitmap graphics mode, change color mode of a bitmap screen and/or clears the current bitmap.

SYNTAX:

BITMAP *[colorMode], [backgroundColor], [clear]*
BITMAP FILL *x1, y1* ***TO*** *x2, y2, [plotType], [color]*
BITMAP CLR | **ON** | **OFF**

DESCRIPTION:

Graphics mode is entered with the BITMAP statement. The C64 has two color modes, hires (320 x 200) and multicolor (160 x 200). The available colors for each pixel has different limitations based on the color mode. Hires mode has greater resolution, but fewer color selections. Multicolor has a greater number of colors that can fit into an 8 X 8 square at a cost of losing half of its horizontal resolution. Refer to the MAPCOL statement for more information about setting colors in both modes.

colorMode (0=hires, 1=multicolor) selects the type of bitmap. In hires mode the resolution is 320x200. In multicolor mode the resolution is 160x100. The first dot is at coordinate 0,0.

backgroundColor (0-15) selects the background color of the bitmap screen. See the COLOR statement (page 6) for the list of background colors.

clear (0=no, 1=yes) is used to clear the bitmap pixels and initialize the color.

BITMAP CLR *c*lears the all pixels and initializes color (even when in text mode).
BITMAP ON switches to bitmap mode using current color mode.
BITMAP OFF switches to normal text and color mode.

BITMAP FILL is used to set dots in a rectangle. The coordinates of the two opposite points (*x1, y1*) and (*x2, y2*) define the rectangle to set the dots based on the optional *plot type* and *color* which default to the previous setting. The direction of the fill animation depends on the order of the coordinates.

EXAMPLE:

```
BITMAP CLR    :REM clear the current bitmap
BITMAP 0      :REM switch display to hires bitmap mode
BITMAP 1,0    :REM switch to multicolor bitmap mode with black bkgnd
BITMAP FILL 100,90 TO 200, 110, 2 :REM flip bits inside rectangle
```

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 is an optional 8-bit binary set of boolean values as follows:

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 hires mode; 1-3 in multicolor mode), selects the color for the line. In multicolor mode, the colors 1-3 will correspond to the colors c1 (default white), c2 (default green), c3 (default yellow). The default colors can be changed by using the MAPCOL statement.

EXAMPLE:

```
CIRCLE 159,99,50,40,%11111111,1,2  :REM 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 specified files or all open files.

filenum (0-255) closes the specified file referenced by number. You can close multiple file numbers by separating them with a comma. No error will result if the file is not open.

CLOSE FILES will close all logical files and restore default I/O devices.

EXAMPLE:

```
CLOSE 1       :REM close file 1
CLOSE 1,7,9   :REM close files 1,7 and 9
CLOSE FILES   :REM close all files
```

COLOR

PURPOSE:

To select the color for the screen background, border and printed characters.

SYNTAX:

COLOR *[text], [background], [border]*

DESCRIPTION:

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

background color (0-15) sets the background color.

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

The following is a table of the colors & their numbers:

VALUE	COLOR	VALUE	COLOR
0	black	8	orange
1	white	9	brown
2	red	10	light red
3	cyan	11	gray 1 (dark)
4	purple	12	gray 2 (medium)
5	green	13	light green
6	blue	14	light blue
7	yellow	15	gray 3 (light)

EXAMPLE:

```
COLOR 14,6,14   :REM set commodore standard color scheme
COLOR ,14       :REM set background color to light blue
COLOR 2         :REM select red as the color for the next print
```

DELETE

PURPOSE:

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

SYNTAX:

DELETE
DELETE *line#*
DELETE *start*-*stop*
DELETE *start*-

DESCRIPTION:

DELETE aids the programmer when editing a program. Any line or groups of lines may be removed from the program rapidly.

DELETE used alone does the same task as NEW.

line# is the line to delete.

start is the first line to be deleted. This value may be omitted if the first line to delete is the first line in the program.

stop is the last line to be deleted. This value may be omitted if the last line to delete is the last line in the program.

MDBASIC always returns to direct mode after the DELETE statement is executed because the program is changed & the variables are lost.

EXAMPLE:

```
DELETE 50       :REM delete line 40
DELETE          :REM deletes entire program
DELETE 10-70    :REM deletes lines 10 thru 70
DELETE -100     :REM deletes all lines up to line 100
```

DESIGN

PURPOSE:

To redefine a character's shape or to turn on/off redefined character mode.

SYNTAX:

DESIGN NEW | ON | OFF
DESIGN *screen code, 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 ON turns on redefined character mode allowing the programmer to change any or all of the existing characters in upper & lower case as well as the graphics character set. Bitmap mode and multicolor mode are turned off by default. This mode has its own text screen so upon entry, the screen will consist of the characters that were printed on it, allowing the programmer to have two full screens of text to work with if desired. If this screen has not been used before, most likely garbage is displayed, so it is necessary to clear the screen after executing DESIGN ON. You do not have to have design mode on (visible) to be able to apply new designs.

NEW allows the Commodore character set to be copied for all characters. When used, the current character definitions are replaced with a copy of the Commodore 64 ROM image.

Screen RAM is located from $CC00-$CFE7 (52224-53223) rather than $0400-$07E7 (1024-2023). Any PEEKs and POKEs relating to screen RAM must be in this area for correct operation.

Sprite data space is from $C000-$CBBF (49152-52223) using sprite pointers from 0-47, allowing 48 different sprite shapes.

DESIGN OFF returns the computer to the normal text graphics mode. This task can also be done by using TEXT.

screen code (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:

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.

EXAMPLE:

```
DESIGN NEW                  :REM copy ROM chars to RAM for init design
DESIGN 1,0, 0,0,0,8,8,0,0,0 :REM scan code 1 (ASCII 'A') changes to a dot
DESIGN ON                   :REM switch to redefined char mode screen
SCREEN CLR:PRINT"A"         :REM clear screen and display new char design
DELAY 120                   :REM wait 120 jiffies (2 seconds)
DESIGN OFF                  :REM switch to std text mode screen
```

DISK

PURPOSE:

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

SYNTAX:

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

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 specified variable. The format of this string is as follows:

```
[Status Code],[Message],[Info1],[Info2]
00, OK,00,00
00,FILES SCRATCHED,01,00
```

command$ is a string of characters containing the drive number, function to perform, and the parameters to sent to the drive. The following is the commands offered by the 1541 disk drive:

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.

string$ is an optional string variable to store the DOS response message.

EXAMPLE:

```
DISK"I0"                  :REM initialize drive 0
DISK"N0:JUNK DISK,JD"     :REM format drive 0, name="JUNK DISK", id="JD"
DISK"V"                   :REM validates diskette in drive 0
DISK"S1:POKER"            :REM erases program "POKER" on drive 1
DISK"C1:DATA=0:DATA"      :REM copies "DATA" from drive 0 to drive 1
```

```
DISK"C0:DATA.BAK=0:DATA"  :REM makes backup copy of "DATA" on drive 0
```

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 directions and distances for drawing as follows:

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

The values following these 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 :REM set current coordinates for draw
DRAW "C2,R30,P3,R30,P1,R30,D25,L90,U25" :REM red box with open top
```

DUMP

PURPOSE:

To print various information including a program listing, a text screen, graphics screen, current variables or user-defined expression.

SYNTAX:

DUMP LIST *[start]-[stop]*
DUMP SCREEN
DUMP BITMAP [*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.

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 allows the current text screen to be printed on the printer.

DUMP BITMAP allows the bitmapped screen to be printed on the printer.

size (1 or 2, default=1) selects one of the two sizes to print as follows: 1=normal size, 2=double size

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-             :REM dumps program listing from line 100
DUMP SPC(3)+"PRINTER OK."  :REM simple test for printer
DUMP SCREEN                :REM dumps everything on the text screen
DUMP BITMAP 2              :REM print bitmap 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.

THEN and/or ELSE may be followed by either a line number for branching, or one or more statements to be executed.

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

ELSE will apply to the first IF/THEN statement having a false *expression* on the same program line. Consider the following example:

```
     10 A=1:B=2:C=1
     20 IF A=B THEN IF B=C THEN PRINT"A=C" : ELSE PRINT"A<>C"
```

The output will be:

```
     A<>C
```

EXAMPLE:

```
IF A$="N" THEN PRINT"NO":ELSE PRINT"YES":REM yes or no will be printed
IF X THEN 100 ELSE 200                  :REM goto 100 if x<>0, 200 if x=0
IF X=1 THEN 200 ELSE X = 0              :REM goto 200 if x=1, otherwise set x=0
```

ENVELOPE

PURPOSE:

To set the duration of the volume envelop 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 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 part of the cycle.

voice# (1-3) selects which voice will have 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
8	0.10 seconds
9	0.25 seconds
10	0.50 seconds
11	0.08 seconds
12	1.00 seconds
13	3.00 seconds
14	5.00 seconds
15	8.00 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	VALUE	DURATION
0	0.006 seconds	8	0.30 seconds
1	0.024 seconds	9	0.75 seconds
2	0.048 seconds	10	1.50 seconds
3	0.972 seconds	11	2.40 seconds
4	0.114 seconds	12	3.00 seconds
5	0.168 seconds	13	9.00 seconds
6	0.204 seconds	14	15.00 seconds
7	0.240 seconds	15	24.00 seconds

sustain (0-15) selects the volume level at which the note is sustained. Following the decay cycle, the volume of the output of the voice will remain at the selected sustain level until the release cycle is started using the WAVE statement (page 57).

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 0-15 in the decay chart.

The attack, decay, sustain cycle is started by the WAVE statement (page 57) by setting the gate operand=1. The sound being emitted will stay at the sustain volume until the release cycle is starting with gate=0.

You may notice the volume of the sound does not quite get to 0 at the end of the release cycle, and you may need to turn off the sound to get rid of the residual noise. This can be done by setting the waveform, frequency or volume to zero.

EXAMPLE:

```
ENVELOPE 1,15,15,15,15       :REM maximum values for all settings
ENVELOPE 3,0,0,0,0           :REM clear voice#3's envelope
```

ERROR

PURPOSE:

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

SYNTAX:

ERROR *err#*
ERROR 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
 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 FILENAME
 9 ILLEGAL DEVICE NUMBER
10 NEXT WITHOUT FOR
11 SYNTAX
12 RETURN WITHOUT GOSUB
13 OUT OF DATA
14 ILLEGAL QUANTITY
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 MISSING OPERAND
32 ILLEGAL VOICE NUMBER
33 ILLEGAL SPRITE NUMBER
34 ILLEGAL COORDINATE
35 CAN'T RESUME
36-127 USER DEFINED
```

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

ERROR CLR clears the info from the last error.

EXAMPLE:

```
10 ON ERROR RESUME NEXT: PRINT"HERE WE GO"
20 ERROR 28 :REM RAISE VERIFY ERROR
30 PRINT "LAST ERROR#:";ERROR(0);" LINE#:";ERROR(1)
```

ERROR()

PURPOSE:

To return information about the last error that was trapped.

SYNTAX:

E = **ERROR(*N*)**

DESCRIPTION:

This function returns information about the last error that occurred while error trapping was enabled with ON ERROR GOTO.

N (0 or 1) selects the error information to return. When *N*=0 the error number is returned. When *N*=1 the line number of statement that caused the error is returned.

See the ERROR and ON ERROR statments for more information.

EXAMPLE:

```
ON ERROR GOTO 1000      :REM if any errors occur, goto line 1000
.(main program here)
1000 PRINT ERROR(0)     :REM print error num
1010 RESUME NEXT        :REM ignore all errors, execute next statement
```

EXPAND

PURPOSE:

Double the width and/or height of a sprite.

SYNTAX:

EXPAND *sprite#*
EXPAND *sprite#, [x], [y]*
EXPAND ON | **OFF**

DESCRIPTION:

Sprites can expand their x and/or y size by 2 times their normal size. Whenever an axis has been expanded, each pixel is drawn twice which increases the size of the image. Both the x and y axis are independently expandable.

sprite# (0-7) defines which sprite to expand.

x (1=expand, 0=normal) is a Boolean expression used to expand the x axis or to not expand the x axis. This value may be skipped by using a comma in its place which will not effect the current setting.

y (1=expand, 0=normal) is a Boolean expression used to expand the y axis or to not expand the y axis. This value can be omitted to not effect the current setting.

EXPAND ON or EXPAND OFF will turn ON or OFF x and y expansion for all 8 sprites.

EXAMPLE:

```
EXPAND 0            :REM expand sprite 0 x & y
EXPAND 6,1          :REM expand sprite 6 x, leave y as is
EXPAND 1,,1         :REM expand sprite 1 y, leave x as is
EXPAND 2,1,0        :REM expand sprite 2 x, turn off y expansion
EXPAND 1,0,0        :REM turn off sprite 1 x & y expansion
```

FILES

PURPOSE:

To display the directory of the current diskette.

SYNTAX:

FILES *[volume$] [,device]*

DESCRIPTION:

FILES allows the listing of the directory without loading it into memory. The directory is displayed in the normal directory format.

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

volume$ ("*d*:*volume*") is the search string for the directory, allowing parts of the directory to be displayed instead of the full directory. The volume also allows the reading of multiple drives. *d*: (0 or 1, default=0) is the drive number to read from (refer to your disk drive manual for more information.)

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 stopped by pressing the break key.

EXAMPLE:

```
FILES                  :REM drive 0's files
FILES,9                :REM all files on drive 0, device 9
FILES"1:*",9           :REM all files on drive 1, device 9
FILES"A*"              :REM all files on drive 0 that start with A
FILES"0:A*"            :REM same as above
FILES"1:*"             :REM all files on drive 1
FILES"0:*",9           :REM all files on drive 0, device 9
```

FILL

PURPOSE:

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

SYNTAX:

FILL *tx1, ty1* **TO** *tx2, ty2, [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.

tx1 & *ty1* define the upper left hand corner to start the fill process.

tx2 & *ty2* define the lower right hand corner to end the fill process.

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

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

color (0-255) is the color that is to be filled in the defined area of the screen. Note the value ranges to 255. This is for support of the flashing text feature (see COLOR statement) and multicolor bitmap mode. Changing the upper & lower nibbles of screen RAM & the lower nibble of color RAM causes a section of the multicolor bitmap screen to change color(s). See the Commodore 64 Programmer's Reference Guide pg. 128 for more details).

EXAMPLE:

```
FILL 0,0,39,0,,2      :REM top line changes color to red
FILL 20,0,39,24,32    :REM fill bottom of screen with spaces
```

FILTER

PURPOSE:

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

SYNTAX:

FILTER *[cutoff], [resonance], [type]*
FILTER VOICE *voice#, [boolean*]

DESCRIPTION:

The SID chip allows attenuation (make quieter/louder) of a certain range of frequencies. Any combination of the three voices can be filtered but there is only one filter for all voices.

cutoff (26.2 - 11904.5) selects the center frequency (AKA: center frequency for band pass & band reject filters) in Hertz (Hz) that makes any sounds quieter the further above and/or below this frequency depending on the type of filter used.

resonance (0-15) allows peaking the volume of those frequencies nearest the cutoff, creating an even sharper filtering effect.

type (0-4) selects the type of filter used with the center frequency:
  0=none -disable filter (all voices)
  1=low pass -suppress frequencies above the cutoff frequency
  2=band pass -suppress frequencies above & below the cutoff frequency
  3=high pass -suppress frequencies below the cutoff frequency
  4=band reject -cuts off frequencies nearest the cutoff frequency

High & low pass filters reduce the volume of the frequencies furthest away from the cutoff frequency by 12dB per octave.

The band pass filter attenuates the frequencies nearest to the center frequency by 6dB per octave.

The band reject filter reduces the volume of the frequencies furthest away from the center frequency by 6dB per octave.

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

EXAMPLE:

```
FILTER 1500.5, 15, 1  :REM low pass, cutoff=1500.5Hz
FILTER 2145.5, 0, 4   :REM band reject, center=2145.5Hz
FILTER 500            :REM just change the filter frequency
```

```
FILTER ,0             :REM just turn off resonance
```

FIND

PURPOSE:

To search through a program for statements or sequence of characters.

SYNTAX:

FIND"*string*
FIND*code*

DESCRIPTION:

FIND is used for debugging purposes. When making a large program, it is sometimes necessary to search through it to find things like what strings have been used, statements that have been used, or a miscellaneous string of characters.

"string is used when searching for ASCII text (not commands). The ending quotation mark is left off unless it is part of the string that is to be searched for.

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

When FIND makes a match, the line is displayed and the search process continues until the end of the program is reached or when the BREAK key is pressed.

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:

```
FINDA$=             (finds all 'A$=')
FINDGOSUB 200       (finds all 'GOSUB 200')
FIND"THE            (finds all 'THE')
FINDPRINTA*B        (finds all 'PRINTA*B')
```

HEX$()

PURPOSE:

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

SYNTAX:

*v$* = **HEX$(***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 is a floating point number then the fractional portion is ignored.

NOTE: To convert binary to hex, use the percent sign (%) then the binary number to represent *n*.

EXAMPLE:

```
PRINT HEX$(49152):  :REM prints "C000"
V$=HEX$(16)         :REM assigns v$ to "10"
PRINT HEX$(%10010)  :REM prints "12"
```

INF()

PURPOSE:

To return various system information.

SYNTAX:

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

DESCRIPTION:

n (0-23) 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 the programmer does not have to concern themselves with which memory locations hold the data. It also makes the program more readable.

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 mode 0=off, 1=on
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	Insert character count
8	Number of open files (max 10)
9	Number of chars in keyboard buffer
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 for text
14	PAL or NTSC video system, 0=NTSC, 1=PAL
15	Kernal Version System 170 v1 C64 0 v2 C64 3 v3 C64 67 SX C64SX 100 4064 PET 64/Educator
16	Current string index of the note playing with PLAY statement
17	Current octave of the note playing with PLAY statement
18	Last plotted X coordinate
19	Last plotted Y coordinate
20	Last plotted coordinate pixel state (0=pixel off, 1=pixel on)
21	Current 16K VIC-II memory bank (0-3)
22	Current BASIC program line number

```
23       Current DATA statement line number
```

EXAMPLE:

```
REM IF ANY OPEN FILES THEN DISPLAY CLOSE MESSAGE AND CLOSE ALL FILES
IF INF(8) > 0 THEN PRINT"CLOSING ALL FILES”:CLOSE ALL

REM PRINT TIME AT TOP OF SCREEN AND RETURN CURSOR TO ORIGINAL POSITION
R=INF(2):C=INF(0) :REM REMEMBER CURRENT CURSOR PHYSICAL POSITION
LOCATE32,0:PRINT TIME$;:LOCATE R,C

X=INF(18):Y=INF(19):P=INF(20) :REM GET LAST PLOT INFO
PLOT 160,100,3 :REM SET CURRENT COORDINATE WITHOUT PLOTTING A DOT
PRINT"COORDINATE: (";INF(18);",";INF(19);") HAS PIXEL VALUE: ";INF(20)


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


0 DIMK$(4)
1 K$(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) :REM KERNAL SYSTEM ID
15 IFK=170THENI=0:ELSEIFK=0THENI=1:ELSEIFK=3THENI=2:ELSEIFK=67THENI=3:ELSEI=4
20 PRINT"THIS COMPUTER SYSTEM:"
30 PRINT"ID  VER  SYSTEM"
40 PRINTK$(I)
```

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:

v = **JOY(***port***)**

DESCRIPTION:

port (0-1) selects the input port that the joystick is plugged in.

v 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. The number returned corresponds to a direction in the table below. Add 128 to the value to:

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

TIP: Use the ON statement with the JOY function allows quick joystick translation.

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 *key#, assign$*
KEY *string$*
KEY LIST | OFF | CLR
KEY WAIT *[variable$]*

DESCRIPTION:

This statement has various forms and uses in both direct and immediate mode. Function key assignments can be changed or displayed but have no effect in immediate mode.

key# (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 typed the keys themselves.

KEY LIST displays the current key assignments.

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

KEY CLR clears the keyboard buffer.

KEY WAIT will wait for a key to appear in the keyboard buffer. An optional string *variable$* can be provided to GET the key's ASCII value.

EXAMPLE:

```
KEY 1,CHR$(147)+"LIST"+CHR$(13) :REM clears screen & lists program
KEY "N":INPUT A$                :REM input with default user response
KEY CLR:KEY WAIT A$:PRINT A$    :REM wait for keypress then print it
```

KEY (function)

PURPOSE:

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

SYNTAX:

K = **KEY**

DESCRIPTION:

This function is used in conjunction with the ON KEY GOSUB statement to return the ASCII value of the key that was pressed. Since there are no parameters the syntax is similar to a variable. The value will be retained until the next keystroke. 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.

EXAMPLE:

```
10 LOCATE,,1 :REM TURN CURSOR ON
20 PRINT">";
30 ON KEY GOSUB 100
40 REM **YOUR MAIN PROGRAM LOOP STARTS HERE**
50 WAIT 5 :REM SIMULATE RUNNING PRG
60 GOTO 40
70 REM **NO KEY TRAP SECTION OF PRG GOES HERE**
80 LOCATE,,0 :REM TURN CURSOR OFF
85 PRINT:PRINT”DONE.”
90 END
100 IF KEY = 32 THEN KEY OFF : RETURN 70
110 PRINT CHR$(KEY);
120 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.

x (0-319), y (0-199) in hi-resolution mode. x (0-159), y (0-99) in multicolor mode.

An ILLEGAL QUANITY ERROR will result if the values exceed the ranges in the current color mode.

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, 0-3 in multicolor mode), sets the color the line will have. In multicolor mode, the colors 1-3 will correspond to the colors c1-c3, selected in the MULTI BITMAP or MAPCOL statement. If *color* = 0 the line will be drawn with the color of the background which is selected using the COLOR statement.

EXAMPLE:

```
LINE 0,0 TO 319,199     :REM draws diagonal line across screen
LINE 160,100 TO 0,199,2 :REM flips line of dots from center to lower left
```

LINE INPUT

PURPOSE:

To input a string of characters from the keyboard or file.

SYNTAX:

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

DESCRIPTION:

LINE INPUT is much like the INPUT statement, except LINE INPUT allows the input of any characters (except return key). This is useful when the string to be entered might have quotations & commas, or the programmer does not want a question mark displayed after, or for the prompt.

"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. When input comes from the keyboard it is limited to 80 characters. When input comes from a file (LINE INPUT#) the limit is 255.

filenum is the open file to read characters from instead of the keyboard.

EXAMPLE:

```
LINE INPUT A$           :REM no prompt, one string
LINE INPUT A$,B$        :REM no prompt, then enter 2 strings
LINE INPUT"→";A$,B$     :REM prompt once, then input 2 strings
LINE INPUT"->";A$       :REM displays "->" as prompt
LINE INPUT#1, A$        :REM read from open file 1

10 REM***READ FROM FILE***
20 OPEN 1,8,0, "MYFILE.SEQ"
30 LINE INPUT #1, A$
40 PRINT A$
50 IF ST AND 64 GOTO 70
60 GOTO 30
70 CLOSE 1
```

```
80 END
```

LOCATE

PURPOSE:

To place the cursor anywhere on the text screen for printing. Optional parameters allow the cursor to turn on/off during program execution.

SYNTAX:

LOCATE *[x], [y], [cursor]*

DESCRIPTION:

LOCATE is mainly used in conjunction with the PRINT statement. It allows the cursor to be placed anywhere on the screen using an x, y coordinate scheme. It is very useful for text manipulation.

x (0-39) represents the column the cursor moves to. If x exceeds its range, an ILLEGAL COORDINATE ERROR will result.

y (0-24) represents the line the cursor moves to. If y exceed its range, an ILLEGAL COORDINATE ERROR will result.

cursor is a Boolean expression (1 or 0) that will turn the cursor on/off while in program mode.

NOTE: Truncating or using commas allows any one or a combination of these operands to be effected, therefore keeping the other previous values.

EXAMPLE:

```
LOCATE 0,0          :REM cursor at top right corner
LOCATE 15           :REM moves cursor to column 15
LOCATE ,10          :REM moves cursor to line 10
LOCATE ,,1          :REM turn cursor on
```

MAPCOL

PURPOSE:

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

SYNTAX:

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

DESCRIPTION:

MAPCOL is used with bitmap graphics for setting the colors used when plotting dots.

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. The next plotted dot by any graphics statements will use the new applied setting.

c1 (0-15) changes the default color for plotting dots.

c2 (0-15) changes the default background color of the 8 x 8 square where the plotting dot is located.

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

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

Multicolor mode applies the colors c1, c2 and c3 to the following bit patterns:

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

EXAMPLE:

```
MAPCOL 1        :REM only change plotted dot color
MAPCOL ,2       :REM only change 8x8 square's background color
MAPCOL 1,2      :REM change dot color and 8x8 square's background color
MAPCOL 5,7,10   :REM all colors that will reside in 8x8 square (multicolor)
```

MERGE

PURPOSE:

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

SYNTAX:

MERGE *filename$, device*

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 the two programs so when they come together, the line numbers are not duplicated, and the merging programs start line number is higher than the last line number of the first program that is in memory. This is not always necessary but is a safe practice.

filename$ is the file name of the program that is to be connected to the end of the program in RAM memory.

device is the device number that the merging program will come from.

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

EXAMPLE:

```
MERGE"SUBROUTINE",8     :REM program "SUBROUTINE" is merged with existing program
in memory from the disk drive.
MERGE"PROG2"            :REM program "PROG2" is merged with existing program in
memory from the tape drive.
```

MOVE

PURPOSE:

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

SYNTAX:

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

DESCRIPTION:

Any of the eight sprites can be located anywhere on the screen or move from one point to another at a selected speed. MDBASIC can MOVE a sprite many times faster than regular CBM BASIC.

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

x (0-511) & y (0-255) select the coordinates for the sprites upper left corner. These coordinates are not like bitmap coordinates because the lowest and highest ranges locate the sprite off screen.

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.

EXAMPLE:

```
MOVE 0,24                 :REM only set x coordinate of sprite 0 to 24
MOVE 0,,100               :REM only set y coordinate of sprite 0 to 100
MOVE 1,180,120            :REM locates sprite 1 to center of screen
MOVE 0,24,50 TO 320,229   :REM moves from top-left to bottom right fast
MOVE 7 TO 180,100, 50     :REM moves sprite 7 from current to center screen
```

MULTI

PURPOSE:

Turn on multicolor mode for background or foreground (text) with color selection. Also sets colors to be used by sprites that have multicolor mode enabled.

SYNTAX:

MULTI [TEXT] [*cc1], [cc2]*
MULTI COLOR [*eb1], [eb2], [eb3]*
MULTI SPRITE [*sc1], [sc2]*

DESCRIPTION:

This statement is used to select multicolor mode and to set the colors, except in the SPRITE statement where the colors are just set because there are eight sprites and each individual sprite can have hires or multicolor mode, which is selected in the SPRITE statement. Valid color values range from 0 to 15.

MULTI TEXT enables multi color text mode. This increases the number of background colors available to a character by reducing the number of pixels to display the text. Each pixel is represented by 2 bits. The bit pattern determines the color source. Colors *cc1* & *cc2* define the first two colors that can occupy the 4 x 8 square. The table below lists the four 2-bit patterns and the associated color:

PIXEL	COLOR	SOURCE
00	background color	background color reg 0 (4-bit value)
01	*cc1*	background color reg 1 (4-bit value)
10	*cc2*	background color reg 2 (4-bit value)
11	foreground color	text color RAM byte (3-bit value)

If the character you want to display is not included in scan codes 0-63 then you will need to use the DESIGN command to define your own custom shape. In the following custom character example,
AA=cc1, BB=cc2, CC=foreground color, DD=background color:

IMAGE	BIT PATTERN	DATA
DDAAAADD	00 01 01 00	20
AABBBBAA	01 10 10 01	105
AADDDDAA	01 00 00 01	65
AAAAAAAA	01 01 01 01	85
AADDDDAA	01 00 00 01	65
AACCCCAA	01 11 11 01	125
DDAAAADD	00 01 01 00	20
DDDDDDDD	00 00 00 00	0

This data will make a ball, with multicolor stripes.

If a character is printed to the screen using colors 0-7, the character will be in multicolor. If it is printed using colors 8-15, the character is printed in hi-resolution. This allows multicolor & hi-resolution to be displayed on the same screen at the same time, at expense of loosing colors 0-7 for hires mode. This color can be set by using the COLOR statement.

While in multicolor text mode, if the colors are changed using MULTI TEXT, all the characters on the screen will change color accordingly, producing a special effect.

MULTI COLOR enables extended background color mode. This mode lets you select the background color of each text character, as well as its foreground color. This 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. The next three groups of characters (64-127) select a different background color, but still display the codes 0-63.

BITS	**CODE**	**COLOR**	**SOURCE**
00	0-63		background color reg 0 (4-bit value)
01	64-127	*eb1*	background color reg 1 (4-bit value)
10	128-191	*eb2*	background color reg 2 (4-bit value)
11	192-255	*eb3*	background color reg 3 (4-bit value)

Extended background color mode can be a very useful enhancement for your text displays. It allows the creation of windows, which, because of their different background colors make different bodies of text stand out. The background colors of these windows can be changed using MULTI COLOR, which instantly changes the background of the windows. This is useful for making a particular section of the screen stand out from the rest.

MULTI SPRITE colors *sc1* & *sc2* define the first two colors that can occupy a 24 x 21 sprite. The table below lists the four 2-bit patterns and the associated color:

BIT PAIR	**COLOR TO USE**
00	background color
01	*sc1*
10	*sc2*
11	sprite color

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

EXAMPLE:

```
MULTI COLOR 2,6,7     :REM selects extended background color mode & sets colors
MULTI SPRITE 5,6      :REM changes colors for all sprites bit patterns 01 & 11
MULTI TEXT 15,13      :REM selects multicolor text mode & sets colors
MUTLI 1,2             :REM selects multicolor text mode & sets colors
```

NEW

PURPOSE:

To reset BASIC memory discarding the program or to reset the computer.

SYNTAX:

NEW
NEW SYS

DESCRIPTION:

NEW is the CBM BASIC statement 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 statement.

EXAMPLE:

```
NEW          :REM this will remove the BASIC program
NEWSYS       :REM this will perform a warm restart of the computer
```

OLD

PURPOSE:

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

SYNTAX:

OLD

DESCRIPTION:

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

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

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

A variable was assigned
Another program was loaded
A POKE statement corrupted memory

EXAMPLE:

```
NEW      :REM program gone
LIST     :REM verify program is gone
OLD      :REM program restored
LIST     :REM just to verify it is restored
NEWSYS   :REM computer resets
OLD      :REM program restored
LIST     :REM just to verify it is restored
```

ON ERROR

PURPOSE:

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

SYNTAX:

ON ERROR GOTO *line#*
ON ERROR RESUME NEXT
[ON] ERROR OFF

DESCRIPTION:

ON ERROR 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.

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

ON ERROR RESUME NEXT will ignore all errors and skip to the next statement. The last error number and line is available using the ERROR() function. To clear the last error information use the statement ERROR CLR.

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

E = ERROR(0) returns the number of the error that occurred. This must be used to decode which error occurred (see Appendix B).

E = ERROR(1) 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 ERROR RESUME NEXT     :REM IGNORE ALL ERRORS
20 PRINT 10/INT(RND(TI)*10) :REM POSSIBLE DIVIDE BY ZERO
30 IF ERROR(0) > 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:

```
10 ON KEY GOSUB 1000   :REM when a key is struck, gosub 1000
.
.(main program here)
.
999 END
1000 PRINT KEY(0)      :REM display ASCII of char from keyboard
1010 RETURN            :REM return from subroutine
```

PAINT

PURPOSE:

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

SYNTAX:

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

DESCRIPTION:

PAINT is used in conjunction with the BITMAP statement, although to paint on a bitmap screen, you do not have to be in bitmap graphics. Note that the plot type is omitted here. This is because PAINT works on boundaries made by bits that are on.

When an area is enclosed, the bits that make up the inside of the enclosure can be filled with plotted dots by selecting a start point to paint.

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

plotType (0-3) determines how the shape will be painted. 0=dots off, 1=dots on, 2=flip pixel (on=off, off=on), 3=none (useful with PLOT and DRAW statements).

color (0-15 in hi-resolution mode, 0-3 in multicolor mode), sets the color the dot will have. See the MULTI statement using BITMAP, for explanation of the colors for both modes.

EXAMPLE:

```
PAINT 160,100          :REM paint section starting at center of screen
PAINT 0,0,0            :REM erase dots enclosed in off dots
PAINT 300,10,2         :REM flip dots
```

PLAY

PURPOSE:

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

SYNTAX:

PLAY *S$*
PLAY STOP

DESCRIPTION:

PLAY allows any of the three voices to play real musical notes at eight different octaves in the background. If the volume is currently set to 0 then it will automatically be set to max (15).

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

There are 12 unique notes in each octave: (A#, A, B, C, C#, D, D#, E, F, F#, G, G#). Sharps (#) 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 in the current octave for length *n jiffies*
L*n*	Length *n* (0-99, default 30) of a note in jiffies
O*n*	Octave *n* (0-8, default 4) with 0 and 8 very limited
P[*n*]	Pause play for length *n (0-99, default 30) jiffies*
V*n*	Voice *n* (1-3, default 1) selects the voice to play the notes
W*n*	Waveform *n* (0-8, default 4 @ 50% duty) changes the waveform*

Any invalid characters or values will be ignored with no effect to the sound.
*See the WAVE statement for the list of available waveforms.

PLAY STOP will abort the currently playing notes running in background mode.

EXAMPLE:

```
REM PLAY NOTES IN BACKGROUND WITH TRIANGLE WAVEFORM, OCTAVE 4 (DEFAULT) THEN 5
```

```
PLAY "W1 A-AA#10BCCP#DD#EFF#GG# > A-AA#10BCCP#DD#EFF#GG#"
```

PLAY SPRITE

PURPOSE:

To enable 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:

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:

PLOTTYPE	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 the line will have. In multicolor mode, the colors 1-3 will correspond to the colors c1, c2, c3 in the MAPCOL statement. See the MAPCOL statement for more information.

EXAMPLE:

```
PLOT 160,100       :REM plot center of hires screen
PLOT 80,50,1,1     :REM plot center of multicolor screen with white dot
PLOT 0,0,0         :REM turn off pixel in top left corner
PLOT 319,199,2     :REM flip bottom right corner pixel's condition
```

POKE

PURPOSE:

To write a value to one or more memory addresses.

SYNTAX:

POKE *mem1, value*
POKE *mem1* **TO** *mem2, value [,operation]*

DESCRIPTION:

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

mem1 (0-65535) is the start address of a memory to write.

mem2 (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 an optional operation for writing the value. When omitted the default operation is 0 which simply writes the value to memory. The operation is performed on the current value at each memory address with the *value* parameter.

EXAMPLE:

```
POKE 1024,1          :REM write the value 1 to mem address 1024
POKE 832 TO 896,0    :REM write the value 0 to mem addresses 832-896
POKE $0400 TO $07E7, %10000000, 3 :REM flip bit 7 of all chars on screen

0 REM ***BLINKING TEXT EXAMPLE***
10 LOCATE 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:

v = **POT(***paddle#***)**

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.

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

Input port#0 monitors paddles 1 & 2, input port#1 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 examples below.

NOTE: POT is a function and cannot be used as a statement.

EXAMPLE:

```
10 SPRITE 0,1,1,0,13         :REM turn on sprite 0, white, data ptr 13
15 POKE 832 TO 832+63,255    :REM fill sprite data with all bits on
20 MOVE0,24,50               :REM locate sprite in visible area of screen
25 P1 = POT(1)               :REM capture pot#1's position
30 MOVE 0, (P1 AND 255)+24   :REM sprite#0 x changes with pot#1's position
35 REM if button pressed change sprite color randomly, short pause
40 IF P1> 255 THEN SPRITE0,,RND(TI)*15: WAIT10
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 value.

Seven bytes of memory are allocated for each variable. The first two bytes are used for the variable name, which consists of the ASCII value of the first two letters of the variable name. If the variable name is a single letter, the second byte will contain a zero.

The seventh bit of one or both of these bytes can be set (which would add 128 to the ASCII value of the letter). This indicates the variable type. If neither byte has the seventh bit set, the variable is the regular floating point type. If only the first byte has its seventh bit set, the variable is a string. If only the second byte
has its seventh bit set, the variable is a defined function (FN). If both bytes have the seventh bit set, the variable is an integer.

The use of the other five bytes depends on the type of variable. A floating point variable will use the five bytes to store the value of the variable in floating point format. An integer will have its value stored in the third and fourth bytes, high byte first, and the other three will be unused.

A string variable will use the third byte for its length, and the fourth and fifth bytes for a pointer to the address of the string text, leaving the last two bytes unused. Note that 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 ($33).

A function definition will use the third and fourth bytes for a pointer to the address in the BASIC program text where the function definition starts. It uses the fifth and sixth bytes for a pointer to the dependent variable (the X of FN A(X)). The final byte is not used.

EXAMPLE:

```
P = PTR(A$)       :REM P HOLDS THE ADDRESS OF THE STRING DESCRIPTOR OF A$
```

PULSE

PURPOSE:

Sets the pulse value for any of the three voices.

SYNTAX:

PULSE *voice#, width*

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 set the pulse frequency.

width (0-100) determines the duty cycle. This value is expressed in percent (%) 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 with the pulse waveform.

NOTE: In order for this statement to affect the sound, a pulse waveform must be set for the corresponding voice by using the WAVE statement.

EXAMPLE:

```
PULSE 1,100       :REM voice#1 stays high continuously
PULSE 3,33.3333   :REM voice#3 will stay high 1/3rd of the time
PULSE 1,50        :REM voice#1 has a square wave output
```

RENUM

PURPOSE:

To renumber a program with a selected increment.

SYNTAX:

RENUM *[start], [increment]*

DESCRIPTION:

RENUM allows a programmer to space the number of lines between each program line allowing easy insertion of programming anywhere in the program. It also makes a program look neat and easy to follow.

start is the optional (default 10) new start line that the program to renumber will begin with.

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.

EXAMPLE:

```
RENUM            :REM program starts with line 10 with 10 increments
RENUM 1000,10    :REM program starts with line 1000 with 10 increments
RENUM 100        :REM program starts with line 100 with 100 increments
```

RESTORE

PURPOSE:

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

SYNTAX:

RESTORE
RESTORE *[line#]*

DESCRIPTION:

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

RESTORE is an existing CBM BASIC statement which resets the data pointer to the first data line in the program. RESTORE has been augmented to allow resetting to a specific line in the program.

line# is an optional parameter specifying the line where the next READ statement will get DATA. If line# is not part of the program, an UNDEF'D STATEMENT will occur. If line# does not have a DATA statement, READ 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         :REM sets data pointer at first line in program
RESTORE 100     :REM data is found starting at line 100
```

RESUME

PURPOSE:

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

SYNTAX:

RESUME
RESUME NEXT
RESUME *[line#]*

DESCRIPTION:

RESUME is used in conjunction with the ON ERROR 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 *line#* redirects the program execution to any line in the program. Resuming to an invalid *line#* will cause an UNDEF'D STATEMENT ERROR.

Regardless of how resuming is done, the previous error number is set to 0 and the line number is set to -1.

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 ERROR 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 ERROR GOTO. If executed without an error trapped by ON ERROR GOTO line# then CAN'T RESUME ERROR will occur. Resuming to an invalid *line#* will cause an UNDEF'D STATEMENT ERROR.

EXAMPLE:

```
10 ON ERROR GOTO 1000
…(main program here)…
999 END
1000 PRINT"ERR:";ERROR(0);"LINE:";ERROR(1)
```

```
1001 RESUME NEXT:REM SKIP OVER STATEMENT THAT CAUSED THE ERROR
```

RETURN

PURPOSE:

To return from a subroutine called by GOSUB.

SYNTAX:

RETURN
RETURN *[line#]*

DESCRIPTION:

RETURN is a CBM BASIC statement used to return to the statement after the GOSUB statement that called the function. RETURN has been augmented by MDBASIC to enable overriding the return location in the program.

RETURN *line#* will abort the GOSUB call and discard the stack information about where to return. The RETURN statement will then perform a GOTO with the line number provided.

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 dependent on line numbers on every line.

The FOR/NEXT statement block is the only control-flow statement which cannot break out of the loop without reaching the end case tested by NEXT. Any attempt to GOTO a line outside of the loop would leave information on the stack that could eventually result in a STACK OVERFLOW ERROR.

If RETURN is executed without a GOSUB, RETURN WITHOUT GOSUB ERROR will occur.

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 the desired precision.

SYNTAX:

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

DESCRIPTION:

ROUND is a numeric function that returns the given value rounded to a specified number of decimal places.

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

precision (optional) is number of digits of precision. If no precision is specified the default is 0, which will return the nearest whole number value. If precision is negative, the whole number portion will be affected.

EXAMPLE:

```
PRINT ROUND(5.0125)    :REM RESULT IS 5
PRINT ROUND(5.0125,2)  :REM RESULT IS 5.02
PRINT ROUND(125.23,-1) :REM RESULT IS 130
```

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 is the device number that the information will come from.
If no device is selected, the default is 1 (tape drive).

RUN will perform the following initializations:
1. Clear all variables (CLR)
2. Close all open files (CLOSE FILES)
3. Turn off all MDBASIC background processes (PLAY STOP, PLAY SPRITE OFF)
4. Turn off key trapping (KEY OFF)
5. Turn off error trapping (ERROR OFF)
6. Turn off sound by clearing all SID registers (VOICE CLR)

RUN will not affect the current state of the display. Any text on the screen, color settings, graphics mode and sprite settings, etc. will remain. Each program should initialize these settings as needed. This allows multiple programs to chain together and act as a single program.

EXAMPLE:

```
RUN              :REM execute program at first line#
RUN"PROGAM2",8   :REM loads & executes "program2" from disk
RUN"MENU"        :REM loads & executes "menu" from tape
```

SAVE

PURPOSE:

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

SYNTAX:

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 called a “binary save”. The first syntax listed above is the standard CBM BASIC syntax which always uses the start address of BASIC program text ($0800). The second syntax is the augmented SAVE which enables 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.

device (default 1) is the device number for writing the bytes.

secondary is the secondary address.

Refer to the Commodore 64 User’s manual for more details.

EXAMPLE:

```
SAVE $C000, $CFFF, “ML_PRG”,8   :REM SAVE 4K BYTES IN HIRAM
SAVE 1024, 2023, “SCREEN”, 8    :REM SAVE TEXT SCREEN
SAVE “MYFILE”                   :REM SAVE BASIC PRG TO TAPE WITH NO NAME
```

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, default 8) is the number of bits that make up the packet for one "word" of data. The possible values are 5, 6, 7 or 8 otherwise an ILLEGAL QUANTITY ERROR will result. Some data, like ASCII characters do not need all 8 bits so it is better to reduce the size to increase overall throughput.

stopbits is the number of bits used to provide markers in the transmission to demark the end of a packet of data.

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) 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 F for a more detailed example.

EXAMPLE:

```
SERIAL OPEN              :REM open rs-232 channel with use defaults
```

```
SERIAL WAIT 300 READ S$   :REM read a string of bytes with 5 sec timeout
SERIAL PRINT A$;          :REM write a string without carriage return
SERIAL CLOSE              :REM close the channel
```

SCREEN

PURPOSE:

To turn on/off the screen or to shrink the visible display area.

SYNTAX:

SCREEN ON
SCREEN OFF
SCREEN CLR
SCREEN *[columns], [rows]*

DESCRIPTION:

The screen statement is used when the side edges and/or the bottom edges need to be hidden from view. Sprite graphics & fine scrolling text on the 64 commonly use this feature to hide unwanted display. For example, an enlarged sprite will stick out of the border at its zero axis. Also garbage will be seen when using fine scrolling to center text.

SCREEN ON & SCREEN OFF enables/disables the screen output. The color of the full screen will be the color of the border.

SCREEN CLR will clear the text screen. This is the same as PRINT CHR$(147).

columns (38 or 40) selects the amount of columns the screen will have. When selecting 38 columns, the screen will visually have 40 columns but the other two are just covered and can be printed to. If *columns* is not to be affected, use a comma to skip over it.

rows (23 or 25) selects the amount of rows the screen will have. When selecting 23 rows, the screen will visually have 25 rows but the other two are just covered and can be printed to.

NOTE: SCREEN OFF increases programming speed. It also allows programs to be correctly loaded from the old 1540 disk drives.

EXAMPLE:

```
SCREEN OFF         :REM turn output to screen off
SCREEN ,25         :REM select 25 rows
```

SCROLL

PURPOSE:

Moves a section of text in a selected direction.

SYNTAX:

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

DESCRIPTION:

SCROLL allows any section of the screen to be scrolled up, down, left, or right. This statement has many uses an produces dramatic displays.

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) 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    :REM scroll whole screen up, no wrapping
SCROLL 0,0 TO 39,0,3   :REM scroll top line right, with wrapping
```

SPRITE

PURPOSE:

To set/change the configuration of any of the eight sprites. The configuration consists of the visibility, color & color mode, foreground priority, and data pointer for the image.

SYNTAX:

SPRITE *sprite#, [visible], [color], [multi], [data pointer], [priority]*
SPRITE ON | **OFF**

DESCRIPTION:

A sprite is a Movable Object Block (MOB) that is 24 bits wide and 21 bits in height, using 64 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 using the MOVE statement.

sprite# (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 (0-15) selects the color for the sprite. Refer to the table on page 11.

multi (0 or 1) is a Boolean expression that puts the selected sprite in multicolor mode or high resolution. 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 allow this feature. See the MULTI SPRITE statement for more info.

data pointer (0-255) is the index of the 64-byte data block being used to define the shape of a sprite. The memory address can be calculated by the formula:

ADDRESS=(DATA POINTER*64)+(BANK*16384)

BANK (0-3) is the 16K VIC-II base video memory bank being used. MDBASIC selects the BANK depending on the 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	3	49152-65535	4K HIRAM, 4K DEVICE RAM, 16K KERNEL ROM

& Custom Text*			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.

To design a sprite shape, you need 64 bytes of data that represents the sprite and put it into memory starting at the address calculated with the formula on the prior page. 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

These 63 numbers make up the sprite shape. A sprite is three bytes wide and 21 bytes tall. The data can easily be put into DATA statements or a sequential file for a loop to READ and POKE them into memory.

SPRITE ON or SPRITE OFF will turn all 8 sprites ON or OFF.

Since sprite memory is represented in 64-byte blocks you may want to add an additional byte to the end of your data set however it is not used. You could use the byte 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 be encoded the bits of the 64th byte which you can apply to the sprite using separate statements.

EXAMPLE:

```
SPRITE 0,1,7,0,13  :REM sprite#0 on, yellow, no multicolor, pointer#13
SPRITE 0,,,,14     :REM change sprite 0's pointer to 14
SPRITE 2,,,,,1     :REM change sprite 2’s priority to be under text/graphics
```

SWAP

PURPOSE:

Exchange the values of two variables.

SYNTAX:

SWAP *variable1, variable2*

DESCRIPTION:

SWAP is very useful in bubble sorts. A bubble sort is a routine that puts numbers or strings in numeric/alphabetic order. SWAP discards the need for a third buffer variable, and reassigning variables, to make the exchange. Reassigning variables eats up free memory space (especially with strings) causing a garbage collection to occur faster than usual. When a garbage collection takes place, the BASIC programming is halted. The computer seems to "lock-up" while it discards variable data that is not used anymore. This process can be very long and is definitely a serious problem.

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 etc. If the variables are of different types, a TYPE MISMATCH ERROR will occur.

EXAMPLE:

```
SWAP A,B          :REM A & B exchange values
SWAP A$(1),A$(2)  :REM A$(1) & A$(2) exchange values
SWAP X%,Y%        :REM X% & Y% exchange values
SWAP N$,K$        :REM N$ & K$ exchange strings
```

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.

```
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,0,10     :REM CALL KERNAL SUBROUTINE TO CLEAR LINE 10
SYS $E9FF,0,INF(2) :REM CALL KERNAL SUBROUTINE TO CLEAR CURRENT LINE
SYS 65511          :REM CALL KERNAL CLALL TO CLOSE ALL FILES
SYS 65520,0,10,5,0 :REM CALL KERNAL PLOT TO LOCATE CURSOR AT ROW 10, COL 5
SYS 49152,0,0,0    :REM CALL USER ML ROUTINE WITH A,X,Y REGS LOADED WITH ZERO
```

TIME & TIME$

PURPOSE:

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

SYNTAX:

TIME CLR
T = **TIME**
T$ = **TIME$**
TIME$ = *string*

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 CBM BASIC's 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 hold the number of seconds since midnight. 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 the variable to store the result of reading the TOD clock.

string 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            :REM get the number of seconds since midnight in var T
PRINT TIME$         :REM print the current time in string format
TIME$ = "13:00:00"  :REM set the time to 1 o'clock pm
```

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 operands, the graphics mode is set to normal text graphics. TEXT also allows the printing of characters anywhere on a bitmap screen with varying sizes.

To return to a graphics screen, the statement used to show the screen must be re executed. For example: BITMAP 0 will redisplay the prior hires bitmap screen.

x (0-319 hires, 0-159 multicolor), y (0-199) are the coordinates on the bitmap of where the top left-hand corner of the first character will print. If a portion of the characters cannot fit on the bitmap then that portion will be truncated.

string$ is the string of characters to be printed on the bitmap screen. There are 29 ASCII control characters supported. Insert is not supported.

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 color the characters will have.

EXAMPLE:

```
TEXT                    :REM return to normal text and color mode.
TEXT 0,0,"CBM",0,1,5    :REM print 'CBM' at top left with tall letters
```

```
TEXT 10,50,"Mark",1     :REM use charset 1 (reverse) with default size
```

TRACE

PURPOSE:

Trace the execution of program statements.

SYNTAX:

TRACE
TRACE *[line#]*

DESCRIPTION:

This statement 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.

line# (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 ERROR 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            :REM run program and trace the lines
TRACE 100        :REM start tracing at line 100
```

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:

Sets 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, or assigning any voice any frequency offered by the SID to any voice. There are 5 basic steps to produce sounds:
1. Set the volume using the VOL statement (page 56).
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 (page 57).

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

frequency (0.0 - 3994.99741 {NTSC} or 3848.597993 {PAL}) 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 may be changed at any time to achieve special effects.

VOICE CLR clears the registers that are used to control the sounds for all three voices. This statement is commonly used at the beginning of a program. When executed, all sounds will be turned completely off, and all sound registers are set to 0.

EXAMPLE:

```
VOICE CLR          :REM initialize SID chip
VOICE 1,2000       :REM voice#1 at 2KHz
```

VOL

PURPOSE:

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

SYNTAX:

VOL *volume*

DESCRIPTION:

This statement is used to turn up/down the volume of all three voices. Selecting separate values for each voice's attack/decay/sustain/release (using the ENVELOPE statement) allows separate volume control of each voice.

volume (0-15) sets the volume of all three voices.

NOTE: No sounds will be heard with any statement unless VOL sets the volume to an audible level.

EXAMPLE:

```
0 REM ** SAMPLE SOUND PROGRAM OF RANDOM STATIC **
10 SOUND CLR           :REM clear all SID registers
20 ENVELOPE 1,0,0,15,0 :REM max sustain volume
30 VOICE 1,2000        :REM set voice#1 at 2KHz
40 WAVE 1,8,1          :REM noise waveform; start ADS cycle
50 VOL 15              :REM volume at max
60 DELAY RND(TI)*25    :REM random delay length
70 VOL 0               :REM turn off noise abruptly
80 DELAY RND(TI)*25    :REM random delay length
90 GOTO 50             :REM loop
```

WAIT

PURPOSE:

To perform a 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 time delay. Waiting on memory can be aborted by simply pressing the run-stop key instead of the combination of the keys run-stop & restore.

The WAIT statement causes program execution to suspend until a given memory address matches a specified bit pattern.

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 time delay including allowing the user to read the screen, a tone to sound, or slowing down the program. CBM BASIC requires the use of a FOR/NEXT loop, which would use a variable. The actual time would have to be calculated through trial & error to see the duration is correct.

EXAMPLE:

```
WAIT 300           :REM wait for 5 seconds
WAIT $DD01,$80     :REM wait till $DD01 is exactly $80
```

```
WAIT $DD01,$80,$7F :REM wait till bit 7 is set in mem $DD01
```

WAVE

PURPOSE:

Set the waveform 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 the envelope cycle. It also is used to synchronize the waveform output with another voice.

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

waveform (0-8) sets the waveform for the voice. 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 determine what part of the envelope to produce and when to start it. The first part of the envelope is the Attack, Decay, Sustain cycle. When gate=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=0, which will start the release cycle. Thus, the gate has two functions: 1=start attack/decay/sustain cycle; 0=start release cycle.

sync (1 or 0, default 0) is a Boolean expression used to synchronize the fundamental frequency of the voice# with another 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 modulation (1 or 0, default 0) is a Boolean expression used to replace the triangle waveform to be replaced with a ring modulated combination of the voice# with another 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 or 0, default 0) is a Boolean expression used to disable the oscillator output of the selected voice. This can be useful when generating very complex waveforms (even for speech synthesis). 1=disable, 0=enable.

EXAMPLE:

```
WAVE 1,1              :REM voc#1 has a triangle waveform
WAVE 3,0,0,0,0,1      :REM disable voice 3
WAVE 3,2,1            :REM voc#3 has a saw tooth waveform; start ADS cycle
```

APPENDIX A

LOADING/SAVING SCREENS

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

This is the normal syntax used when saving/loading programs. The only change is in the secondary address. The range of valid secondary address numbers is 0-31 for serial devices; 32-127 are for other devices.

The secondary address numbers 2, 3 and 4 are used by MDBASIC only. These numbers define which type of screen to save or load. The secondary addresses and the screen they represent are as follows:

```
2 = VIDEO MATRIX       - text and color for current screen being used
3 = CHARSET            - all character shapes from redefined mode
4 = BITMAP             - entire bitmap screen & colors
```

One of these secondary addresses must be used to load or save the appropriate screen. The following are examples of loading & saving.

```
LOAD"SCREEN",8,2  :REM load a text screen into screen RAM
LOAD"FONTS",8,3   :REM save redefined text characters
LOAD"GRAPHIC",8,4 :REM load a bitmap screen
```

NOTE: If a text screen in redefined character mode is saved, only the scan codes (poke codes) on the screen are saved. In order to save the character shapes, they must be saved separately using a secondary address of 3 with a separate filename.

NOTE: When loading/saving a text screen (secondary=2) the data is loaded loaded/saved from the RAM of the current screen video matrix. If in redefined character mode, the screen is at $CC00-$CFE7, if in normal text mode, the screen is at $0400-$07E7.

BINARY SAVE

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

SAVE s*tart, 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           :REM SAVE HIRAM TO TAPE – NO FILENAME
SAVE 49152, 53247,”HIRAM”,8 :REM SAVE HIRAM TO DISK – FILENAME REQUIRED
```

APPENDIX B

ERROR CODES & MESSAGES

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	BAD SPRITE NUMBER
32	MISSING OPERAND
33	BAD VOICE NUMBER
34	ILLEGAL COORDINATE
35	CAN'T RESUME
36-127	USER DEFINED

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

APPENDIX C

DEC/HEX/BIN/ASCII TABLE

DEC	HEX	BINARY	PRINTS
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 SFT/CMDR
9	09	00001001	ENABLE SFT/CMDR
10	0A	00001010	
11	0B	00001011	
12	0C	00001100	
13	0D	00001101	RETURN
14	0E	00001110	LOWER CASE
15	0F	00001111	
16	10	00010000	
17	11	00010001	CRSR DOWN
18	12	00010010	RVS ON
19	13	00010011	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
29	1D	00011101	CRSR RIGHT
30	1E	00011110	GREEN
31	1F	00011111	BLUE
32	20	00100000	SPACE
33	21	00100001	!
34	22	00100010	"
35	23	00100011	#
36	24	00100100	$
37	25	00100101	%

Appendix C (continued)

DEC	HEX	BINARY	PRINTS
38	26	00100110	&
39	27	00100111	'
40	28	00101000	(
41	29	00101001	)
42	2A	00101010	*
43	2B	00101011	+
44	2C	00101100	,
45	2D	00101101	-
46	2E	00101110	.
47	2F	00101111	/
48	30	00110000	1
49	31	00110001	2
50	32	00110010	3
51	33	00110011	4
52	34	00110100	5
53	35	00110101	6
54	36	00110110	7
55	37	00110111	8
56	38	00111000	9
57	39	00111001	:
58	3A	00111010	;
59	3B	00111011	{
60	3C	00111100	?
61	3D	00111101	=
62	3E	00111110	}
63	3F	00111111	?
64	40	01000000	`
65	41	01000001	A
66	42	01000010	B
67	43	01000011	C
68	44	01000100	D
69	45	01000101	E
70	46	01000110	F
71	47	01000111	G
72	48	01001000	H
73	49	01001001	I
74	4A	01001010	J
75	4B	01001011	K
76	4C	01001100	L
77	4D	01001101	M
78	4E	01001110	N

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 quickly 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.

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.

APPENDIX E (continued)

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.

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. A program in assembly language 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.

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.

APPENDIX E (continued)

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.

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.

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 *(Central Processing Unit)*

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.

COBOL

Acronym for **CO**mmon **B**usiness-**O**riented **L**anguage, a computer language suitable for writing complicated business applications programs. It was developed by CODASYL, a committee representing the U.S. Department of Defense, certain computer manufacturers, and major users of data processing equipment. It is designed to express data manipulations and processing problems in English narrative form, in a precise and standard manner.

APPENDIX E (continued)

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.

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 *(Disk Operating System*)

A collection of procedures and techniques that enable the computer to operate using a disk drive system for data entry and storage.

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.

APPENDIX E (continued)

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

A printed copy on paper of computer data.

hardware

The physical equipment that comprises a system.

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.

APPENDIX E (continued)

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

Empty or having no members. This is in contrast to a blank or zero, which indicates the presence of no information. For example, A$="" assigns the variable A$ to a null condition.

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

A quantity or data item involved in an operation. An operand is usually designated by the address portion of an instruction, but it may also be a result, a parameter or an indication of the name or location of the next instruction to be executed.

APPENDIX E (continued)

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.

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.

APPENDIX E (continued)

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.

software

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

syntax

Rules of statement structure in a programming language.

toggle

Alternation of function between two stable states.

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.

APPENDIX F

SAMPLE PROGRAMS

```
10 REM *** MDBASIC SPRITE DEMO ***
20 REM
30 REM  **** BY: MARK BOWREN ****
40 REM
50 PRINT".":SP= 100
55 VOICE CLR
60 VOL15:ENVELOPE1,0,9,0,0:VOICE1,1000
65 MULTI SPRITE 2,6
70 SPRITE0,1,6,1,13:EXPAND0,1,0
75 FORI=832TO832+63 :READA:POKEI,A:NEXT
80 LOCATE7,1:PRINT"*** EXTENDED BASIC V2 ***"
90 LOCATE8,3: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 LOCATE 16,6:PRINT".FOR .THE:."
152 LOCATE17,10:PRINT"COMMODORE 64.":MOVE0 TO 160,120, 120
155 FORI=0TO21:DELAY30:EXPAND0,IAND1,IAND1: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 240,0,170
180 DATA 240,0,168
181 DATA 240,0,160
182 DATA 240,0,128
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 15,248,0
220 DATA 3,224,0
221 DATA 0,0,0
222 DATA 0,0,0,0
```

Appendix E (continued)

```
0 REM *** BITMAP DEMO 1***
5 BITMAP CLR
10 BITMAP1,0:REM MC MODE BLK BKGD
15 MAPCOL 2, 1, 6
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 CIRCLE48,37,23,18,,1,1
32 PAINT48,27,1,1
35 PAINT17,11,1,2
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 DELAY 200
99 TEXT
```

```
0 REM *** BITMAP DEMO 2 ***
10 BITMAP CLR:BITMAP1,0:MAPCOL2,6,15
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 DELAY 200
65 TEXT
```

Appendix E (continued)

```
0 REM SOUND DEMO
1 REM ** DIXIE **
2 REM MUSIC BY: JIM BUTTERFIELD
3 REM
6 W=7
10 PRINT CHR$(147):LOCATE17,0:COLOR,,2
20 PRINT"DIXIE"
30 LOCATE7,2: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 WAVE1,1,0
80 WAVE3,2,0
85 WAVE2,1,0
90 READ S:IF S=0 GOTO 145
95 READ F1,F2,F3
100 MOVE0,(F1/4)+30, (F2/4)+30
105 VOICE1,F1
110 VOICE2,F2
115 VOICE3,F3
120 WAVE1,1,1
125 WAVE3,2,1
130 WAVE2,1,1
135 DELAY S*W:W=7
140 GOTO 75
145 END
150 REM ** 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
```

APPENDIX E (continued)

```
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
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 REM ** 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 REM ***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 REM ***RS-232 SERIAL PORT EXAMPLE 2***
10 REM 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
```

APPENDIX F

BASIC PROGRAMMING GUIDELINES AND SUGGESTIONS FOR PERFORMANCE:

1. Declare all 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 dimensioned variables at the end of the declare section. This will avoid the need for array information to be moved in memory.

3 Declare constants in a variable if they are used heavily 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.

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