Revision A

CBASIC III

128/512K COCO-3 DISK
EXTENDED COLOR BASIC COMPILER
&
PROGRAM EDITING SYSTEM

CER-COMP
5566 RICOCHET AVE.
LAS VEGAS, NEVADA 89110
(702) 452-0632

CBASIC III REFERENCE MANUAL INDEX

CBASIC III Text Editor

CBASIC III Editor Command Section

CBASIC III Compiler Commands

CBASIC III REFERENCE MANUAL INDEX

CBASIC III REFERENCE MANUAL INDEX

APPENDX

Introduction

CBASIC-3 is a complete programming system designed for use on a 128/512K Color Computer 3 with at least one Disk drive. It is completely written in fast efficient Machine Language to take full advantage of the power and flexibility of the 6809E Micro Proccessor and the GIME (Graphics Interrupt Memory Enhancement) chip in the Color Computer 3. It will take full advantage of the 512K of address space available in the Color Computer if 512K is installed, during program Creation, Editing and Compilation. It also provides the user with options to make full use of the 512K available during program run-time.

The Editor contained in CBASIC-3 is used to Create and/or Edit programs for the CBASIC-3 compiler. It is a full featured editor, with functions designed specifically for writing and editing Basic programs. It has built in block Move and Copy functions with automatic program renumbering, easy to use commands for inserting, deleting and overtyping on existing program lines. It is also used for Loading, Saving, Appending and Killing disk files, as well as displaying a disk Directory. Once a program is ready to be compiled, the Editor is issued a command to compile the program, it then calls the compiler portion of the program.

The CBASIC-3 compiler is an optomizing two-pass Basic compiler which converts programs written in Basic to pure 6809 Machine Language programs which are written directly to disk in a LOADM compatible format.

The compiler generated program can be run as a stand-alone RAM based program which may be used without any run-time package. A built in linker/editor automatically selects subroutines from the internal run-time library and inserts one and only one copy of subrountines required directly into the object program. This eliminates the need for cumbersome "run-time" packages that must be loaded separately and usually contain many extra functions not required by the run-time program.

Depending on the specific program, CBASIC-3 can produce programs which may reflect a 5 to 1000 times speed improvement over an interpreter. Since CBASIC-3 also contains statements for supporting Disk and Tape I/O, Hi-Res Graphics and Enhanced Screen formats, it is well suited for a wide range of system programming applications.

NOTE: This entire document was created, edited and printed using a Color Computer III and the TEXTPRO IV - Text Editor & Word Processor, "The Professional Word Processing System".

SCOPE AND REFERENCES

This manual is written to aquaint the user with the features of the CBASIC-3 Editor/Compiler. It should be noted by the user that this is a complex operating system and cannot be fully understood with a single reading. It will require the user many hours of study, usage and experimentation to fully understand the power of this invaluable tool.

It is assumed that the user has a previous knowledge of the Basic Programming Language, as well as a basic understanding of the Tape & Disk Systems of the Color Computer. If this is not the case, you may wish to read the manuals listed below prior to using this manual. This manual is intended as a reference, and is concerned only with describing the additional functions, statements and capabilities provided by the CBASIC-3 Editor/Compiler. It is not the intent or within the scope of this manual to teach the user how to write programs in the Basic or Assembly language.

Radio Shack: "Color Computer 3 EXTENDED Basic"

Radio Shack: "Color Computer Disk System: Owners Manual..

Radio Shack: "TRS-80 Color Computer Assembly Language Programming"

Additional manuals are available from Radio Shack and other sources which describe the Basic Programming Language in general.

Additional Keyboard Characters

CBASIC-3 has several keyboard characters that are not normally available on the CoCo. Some of the additional keys generate the same characters as the arrow & shift keys did previously. The reason for this is, when editing, which uses the arrow and clear keys, you can still generate these key codes if necessary.

New Keyboard Characters

Clear/0 = \ ($5C shift/clear)	Clear/1 = | ($7C *n/a)
Clear/2 = ~ ($7E *n/a)	Clear/3 = [($5B shift/down)
Clear/4 =] ($5D shift/right)	Clear/5 = ^ ($5E up/arrow)
Clear/6 = _ ($5F shift/up)	Clear/7 = ` ($60 *n/a)
Clear/8 = { ($7B *n/a)	Clear/9 = } ($7D *n/a)

STARTUP PROCEDURES

CBASIC-3 is a 6809 machine language program written for use on a Color Computer III with at least 128K of RAM. To Execute the program, place the original disk in your disk drive and enter LOADM"CBASIC3"<enter>. This will cause the program to be loaded into the computers memory and automatically executed. The program will then display an introduction message followed by the amount of free memory available and the "READY" prompt. You are now ready to load a program or enter commands to the CBASIC-3 Text Editor. If an error should occur while trying to load the program, check the disk directory to make sure you are using the same file name as listed in the disk directory. Also make sure you are using the Original disk and not a backup copy.

BACKUP PROCEDURES

Make a backup copy using the "BACKUP" command and put the backup disk in a safe place. **Always use the Original disk to LOAD and Execute the program.** Should the original disk fail, use the "Backup Disk" you created to restore the original disk. The original disk comes recorded on both sides for your added protection against a disk failure. The only way the original disk should be written to is with a "BACKUP" command using the backup disk you created to restore the original.

If you are unable to restore the Original disk due to physical damage etc., return the Original disk only, to Cer-Comp with a check or M.O in the amount of $2.50. We will replace the disk and ship it back to you within 1 working day.

RAMDISK & 512K

If your COCO-3 has 512K of memory installed, CBASIC-3 will automatically install 2 RAMDISKs as drives 2 & 3. These RAMDISKs can be used the same as normal disk drives only they are much faster. You can use then to: save temporary files or Compile programs to just like a normal disk drive. The RAMDISK storage format is compatible with our own RAMDISK program available separately for only $19.95. When using our RAMDISK, files stored in them will be available when you enter or leave CBASIC-3 as well as any of our disk programs.

LIMITED LICENSE TO USERS

Cer-Comp grants you, the owner and original purchaser of CBASIC-3, a limited license for incorporating CBASIC-3 to create your own marketed software products as long as they do not include the use of the HIRES Screen package generated by the compiler using the "HIRES" command. If you wish to use this Proprietary driver in a marketed software product, the author must agree to abide by all of the following conditions:

1. No reproduction of this documentation is permitted.

2. Author or publisher must supply Cer-Comp with a complete copy of the finished software package within 30 days of first publication.

3. The Author or publisher must pay a royality of Five-Dollars ($5.00) for each copy of the program produced, to be paid on a quarterly basis (three month).

Failure to comply with all of the preceeding conditions set forth will result in immediate revocation of limited license and production shall be ceased until all conditions are met to the satisfaction of Cer-Comp. Cer-Comp would, of course appreciate the opportunity to publish any program you develope which incorporates the CBASIC-3 compiler.

DISCLAIMER

A great deal of time and effort was used in the creation of this program, and great care was taken to insure that this program will perform and operate as advertised. If you find a "bug" or problem with this program, please notify us. We will do our best to correct it, but we do not guarantee to do so. Cer-Comp does not warrant the suitability or functioning of its products for any particular user and will not be responsible for damages incidental to its use or misuse. This warranty is in lieu of all other warranties either expressed or implied. Cer-Comp assumes no responsibility for the consequences of the use or misuse of this or any other software and documentation.

Cer-Comp reserves the right to make changes and improvements without prior notice. New revisions will be made available on an exchange basis for a fee of $15.00 to cover the cost of reproduction, manual updates (if required) and return postage.

DISK TEXT EDITOR

DEFINITIONS:

"\" - is the character displayed when the SHIFT & "@" keys are depressed as a delimiter for the 'SEARCH' & 'REPLACE commands. Also see editor command summary.

"()" - items enclosed within these characters are required by that command to perform correctly.

"[]" - items enclosed within these characters are considered as optional, when used they must be in the required order.

"<>" - items enclosed within these characters are comments.

Enter is used to denote an "ENTER" character and is used to signify the completion of a line entry.

"-" - "Dash" is used as a delimiter between line numbers.

"<-" - Left arrow is recognized as a Backspace.

"BREAK - is used for Break control at any time to return to 'READY'. If BREAK is depressed during a line entry or edit, any changes or entries will be ignored.

Any key can be used to stop the present output and it will be resumed upon entry of any key but "BREAK".

All commands can be abbreviated by using the first two characters of the command followed by its normal parameters.

LINE ENTRY:

Enter a line number, followed by a space and text ending with the "Enter" key.

The line buffer is preset to 255 characters and the cursor will not advance past the last character position, nor will it backspace beyond the first character position. Ten characters before the end of line a medium tone beep will be heard and a higher tone beep will be heard at the end of the line. Any time during line entry if an invalid control character key is entered a double low tone beep will be heard.

Entry of a line number over four digits will result in only the last four digits being accepted.

Entry of a line number followed by "Enter" will delete the line previously entered using that line number.

Entry of a new line using a previously entered line number will cause that line to be replaced with the new line.

Entry of a line with a line number between two previously entered line numbers will insert the new line between them.

Printer Requests:

Any time the printer is requested for an operation the status of the printer is checked for ready. If the printer is found to be in a "NOT READY CONDITION", a message to that effect will be displayed and the program will wait for any key on the keyboard to be pressed, except the "BREAK" key. IF the "BREAK" key is depressed the printer output will be aborted. This will allow those users not having a printer to abort an accidental printer request and not hang up the system.

LIST COMMAND

SYNTAX: LIST [line number] (-) [line number]

Entry without line numbers will list the entire file. Entry with a single line number will list only that line. Entry of two line numbers will list from the first line number to the second one. This is very similar the the "Basic" list function.

Example: LIST 100-300<ENTER>

RENUMBER COMMAND

Syntax: RENUMBER [1 digit increment] [starting line #]

Causes the Basic file to be renumbered, if no increment is specified a value of 10 is used. If a starting line # is not specified the increment value is used. If the line #s esceed 9999 before the end of file is reached, the increment value is automatically decreased. The resequence is repeated until a workable value is reached.

Example: RESEQUENCE 5 100
Re-sequence the line numbers in the file begin with '100' and increment each line number by '5'.

DELETE COMMAND:

Syntax: DELETE <begin line#>-<end line#>

The delete function allows large segments of the text buffer to be removed without having to enter each line number to be deleted. If no line specifications are entered the user will be prompted as to whether the entire contents of the buffer are to be deleted. This is mainly to prevent the accidental deletion of the text buffer contents.

Example: DELETE 100-199 <Enter>
Remove all the lines in the text buffer between and including lines 100 thru 199.

SEARCH STRING COMMAND:

Syntax: SEARCH [line #](-)[line #]\[string]\

Searches for all occurrences of the string between the delimiters (Shift @). All the lines containing the specified string will be displayed. If the optional start & stop lines is omitted the search will begin at the beginning of the file to the end of the file. If only the starting line# is specified it will search to the end of file.

Example: SEARCH 100-199 \TEST\
List all the lines containing the string 'TEXT' between lines 100 thru 199.

REPLACE STRING COMMAND:

Syntax: RPLACE [line #](-)[line #] \[string]\[string]\

This function will replace all occurrences of the first string between delimiters (SHIFT @) with the second string. If the optional line #'s are not specified the entire file will be used, if only the starting line # is specified only from there to the end of file will be used, and if both start & end line #'s are specified only the lines including them will be used.

Example: RPLACE 100-999 \TEST\TESTER\
This would tell the editor to replace all occurrences of 'TEST' between lines 100 and 999 with 'TESTER'.

LINE EDIT COMMAND:

Syntax: LEDIT [line #]

Causes the line number specified to be displayed and the cursor to be positioned under the first character of the line. The EDIT mode is then entered, see edit functions under 'AEDIT'.

Example: LEDIT 110 <Enter>
Edit line number 100 using the edit functions.

AUTO EDIT COMMAND:

Syntax: AEDIT [line #]

Causes the automatic edit mode to be entered, if the starting line # is specified the edit function will continue from that line until the end or a cancel edit operation character is entered. All the edit commands are the same as LEDIT (line edit). If no change is required on a line press the Down-Arrow key and the next line will be brought up for editing. If the line is to be deleted just enter Shift"Clear".

Example: AEDIT 100 <Enter>
Begin automatic line editing starting at line 100.

EDIT FUNCTION KEYS

FUNCTION	DEPRESS
MOVE CURSOR RIGHT	Right arrow key
MOVE CURSOR RIGHT 1 WORD	Clear key
MOVE CURSOR LEFT (backspace)	Left arrow key
INSERT SINGLE SPACE	Shift & Up arrow keys
MULTIPLE CHARACTER INSERT on/off	Shift & @ keys
DELETE CHARACTER	Shift & Down arrow keys
MOVE CURSOR TO END OF LINE	Shift & Right arrow keys
MOVE CURSOR TO BEGIN OF LINE	Shift & Left arrow keys
GOTO NEXT SEQUENTIAL LINE	Down arrow key
GOTO PREVIOUS LINE	Up arrow key
END LINE AT CURSOR POSITION	Shift & Clear keys
REPLACE OLD LINE WITH NEW	Enter key
EXIT FROM EDIT MODE	Break key

COPY LINES COMMAND:

Syntax: COPY (from line#)-(to line#) (new location line#)

The copy function allows portion of the current text buffer to be copied to another portion of the file. The lines included in the specifications 'from' and 'to' are copied to the new location line following the destination line. The portion of the file copied is left intact and the file is automatically renumbered upon completion of the copy.

Example: COPY 1100-1345 100
This would place a copy of the lines from 1100 thru 1345 following line 100 .

MOVE & DELETE LINES COMMAND:

Syntax: MOVE (from line#)-(to line#) (new location line#)

The MOVE command works almost exactly the same as the 'COPY' function only the original lines 'from-to' are removed from the file after they are copied to the new location. The file is renumbered the same as in the copy function.

Example: MOVE 1100-1345 100
This would move the lines from 1100 thru 1345 to the next line following line 100.

AUTOMATIC LINE NUMBER COMMAND:

Syntax: AUTO [1 digit increment value] [line #]

Causes the computer to type sequential line numbers incremented by the specified 1 digit value. If not specified the line # will be incremented by 10. Also an optional starting line # can be specified. This is used for entering sequential text lines without having to specify line numbers, they will automatically be typed after each line is entered.

Example: AUTO 100
Enter auto line typing beginning with line '100' with a default increment value of '10'.

MEMORY SIZE COMMAND:

Syntax: SIZE <Enter>

Displays the amount of memory in use, followed by the amount of memory remaining in the text buffer.

PRINTER OUTPUT COMMAND:

Syntax: PRINTER [command line]

Specifies that the next output operation will be output to the printer. Another command may follow the PRINTER command for ease of use. If you want a printed listing of the compiled program, this command must be used prior to the CBASIC-3 command, ex: PR CBASIC-3

Example: PRINTER NLINE LIST<ENTER>
This would tell the editor to list the file to the printer with no line numbers.

EXIT TO BASIC COMMAND:

Syntax: EXIT <Enter>

Causes control to return to 'BASIC'. Once CBASIC-3 is exited you cannot return or re-execute the program, it must be re-loaded from disk.

NEW FILE COMMAND:

Syntax: NEW <Enter>

Causes the memory file buffer to be cleared and all pointers reset to the cold start condition. All previously entered information will be lost. You will be prompted with the message "ARE YOU SURE?", if you enter any character other than a "Y" the command will be ignored.

PRINTER BAUD RATE COMMAND:

Syntax: BRATE <value> Set Printer baud rate

This command will allow users having printers that run at baud rates other than 600 baud, to change printer rates while under CBASIC-3 control. The baud rates are set by entering a value from zero thru seven (0-6) to represent the desired rate. The rate values are as follows: 0=110, 1=300, 2=600, 3=1200, 4=2400, 5=4800, and 6=9600.

Example: >BR 5<enter> Set baud rate to 4800 baud

PRINTER LINE FEED COMMAND:

Syntax: LF<enter> Allow line feed character output

This function is for those users having printers that do not automatically line feed upon receipt of a carriage return character. Normally line feed character output is inhibited, once this command is entered they will be output for each line and cannot be inhibited once enabled.

AUTOMATIC KEY REPEAT DELAY COMMAND:

Syntax: RDELAY <value>

This command allows the user to program whether or not to allow the keyboard keys to automatically repeat and if so, how fast or often it is repeated. If the command is followed by a value of "0" then automatic repeat will be disabled entirely. If a value between 1 and 47 follows, that value will be used to determine how fast the keys will repeat. The smaller the number the faster the key will repeat. The default value is around 15 which causes a repeat at a reasonable rate. Each individual will have to set this to their own personal taste. The delay from the first time a key is pressed until it begins to repeat is approximately 2 seconds and is not adjustable.

Example: RD 5 <enter> Set Repeat Delay to 5 (fast)
RD 0 <enter> Turn Auto Repeat off

SCREEN WIDTH (Characters per line)

Syntax: SW <value>

The SW command allows the user to set the number of characters displayed per line on the Screen. This can be varied from 32 to 80 characters per line in defined steps. The default display comes up in 80 character mode at program startup time, but can be changed to one of 8 different formats. The following values correspond to the number of display characters per line.

1 = 32 (192)	5 = 32 (225)
2 = 40 (192)	6 = 40 (225)
3 = 64 (192)	7 = 64 (225)
4 = 80 (192)	8 = 80 (225)

The numbers in the parenthesis represent the number of vertical scan lines used on the display. The 225 mode gives an extra pixel width between lines so that the decenders on characters will not appear to touch the tops of the letters on the line below. If your TV or Monitor can't handle the extra lines, select one of the 192 line modes.

Example: SW 8 <enter> Set width to 80 chars/line (225)
SW 3 <enter> Set width to 64 chars/line (192)

SCREEN COLOR SELECT:

Syntax: SCREEN <Foreground> <Background>

This command allows the user to select the Foreground (character color) and Background colors for the display. The program defaults to Black characters on a Buff Background (0,63). You can select any color you like from 0 to 63, see page 297 of your COCO-3 manual for some sample color values.

Example: SC 63 0 <enter> BUff chars/Black Background
SC 18 0 <enter> Green chars/Black Background

CHANGE COLOR/MONOCHROME MODE:

Syntax: CColor <enter>

This command allows the user to force the computer to supress the color output to the display or to Enable the color output. By default the program automatically select Monochrome mode when first started up.

Example: CC <enter> Change screen color

DISK FILE SAVE COMMAND:

Syntax: SAVE [file name.extension:disk drive]

The SAVE command writes the file with the specified file name to disk. If no disk drive/id is entered a default drive of "0" is assumed. The file extension is assumed to be a "CBA" file if not specified. The entire file is saved from the text buffer. If the output file is already in use from a previous file that was larger than the text buffer an error message of 'OUTPUT FILE ALREADY IN USE' will be displayed.

Example: SAVE BIOIA.ASM
SAVE BIOIA:3

DISK FILE LOAD COMMAND:

Syntax: LOAD [file name.extension:disk drive]

The LOAD command opens a disk file for input to the text buffer, if line numbers are not included in the text file they will be added. If the file is larger than the available text buffer the user will be prompted for an output file drive and name. If an output file cannot be opened the input file will be closed and only that portion of the file will be accessable for editing. When a duplicate output file is encountered it is automatically removed by the R.S disk system so be aware when specifying file names.

Example: LOAD BIOIA:3

Open the file BIOIA on drive #3 for input and read it into the available text buffer.

DISK FILE APPEND COMMAND:

Syntax: APPEND [file name.extension:disk drive]

The APPEND command adds the file to the end of the present memory file. The Disk drive and file extension options are the same as the'LOAD' command. If the input file is already in use an appropriate error message will be displayed.

DISK DRIVE DEFAULT

Syntax: DRIVE <number>

The Drive command allows you to specify a default disk drive for Disk commands. The value can be in the range of 0 to 65, this allows Hard Disk users to use up to a 10 Meg. drive.

Example: DRIVE 3

DISK DIRECTORY DISPLAY COMMAND:

Syntax: DIR <drive number>

The DIR command allows the user to examine the directory on a specified disk drive. If the drive number is not specified a default drive of "0" is assumed. The disk directory is displayed the same as if the command had been executed from basic and the "Shift @" must be used to pause the display during this command.

Example: DIR 2

This would list the entire directory from the disk on drive number two.

KILL DISK FILE COMMAND:

Syntax: KILL [file name.extension:disk drive]

The KILL command allows you to remove unwanted files from the specified disk. It works basically the same as the Basic "KILL" command except the file extension will automatically default to a "CBA" extension. If not specified the disk drive will automatically default to drive "0". Any errors will be reported the same as normal disk errors.

Example: KILL BIOIA.TXT:3

Remove the file BIOIA.TXT from the disk on drive number 3.

CBASIC COMPILER COMMAND:

Syntax: CBASIC [file name.extension:disk drive]

The CBASIC command is used to compile the Basic program in memory. Optionally a disk file name can be specified for the compiled object program. If no file name is specified a program will not be created, this can be usefull for testing a programs syntax or generating a printed listing only.

Example: PR CBASIC BIOIA:1

This command string would enable output to the printer (PR) and then call the CBASIC compiler, the program would be compiled with the object code file being written to a file labeled "BIOIA" on drive #1, the extension default would be .BIN.

CBASIC-3 PROGRAM STRUCTURE

A CBASIC-3 program consists of a series of "source lines". A source line consists of a line number followed by one or more CBASIC-3 Statements. If the source line contains more than one statement a colon : character is used to separate the statements. A source line may contain up to 250 characters.

Line numbers are decimal numbers which are up to "four" digits and positive. These must appear sequentially in a program and may not be duplicated. When converting a Color Basic program which has line numbers greater than 9999, renumber the program using Color Basic before saving the program to disk in ASCII format.

Spaces in CBASIC-3 statements are not required, however they may be used to improve readability (except when used in string constants or following variable names that preceed a command). Unlike interpreters, REMark statements and spaces do not affect program size or speed and may be used generously to improve program readability and documentation.

Example of program structure:

```
100 PRINT "THIS PROGRAM FINDS THE AVERAGE OF A SERIES OF NUMBERS"
110 INPUT "HOW MANY NUMBERS "; N : T=0
120 FORI=1TON: INPUT "NEXT NUMBER";I : T=T+1 :NEXT I
130 PRINT:PRINT:PRINT "AVERAGE IS";T/N
140 PRINT "DO YOU WANT TO CONTINUE":INPUT A$
150 IF A$="YES" THEN 100 ELSE END
```

As you can see in the sample program, the syntax of a CBASIC-3 program is very similar to that of the Color Basic interpreter. Most of the CBASIC-3 statements are identical in format to Color Basic. Many programs may be written with the interpreter for testing and debugging, then saved to disk in ASCII format, loaded into CBASIC-3 and compiled. Most of the syntax differences between CBASIC-3 and Color Basic can be used in the interpreter for testing and debugging. However, there are some syntax formats in Color Basic that cannot be used in CBASIC-3. These minor differences will become apparent as you use CBASIC-3, and should not pose much of a problem in converting existing Color Basic programs.

ARITHMETIC OPERATIONS

NUMBERS

CBASIC-3's numeric data type is internally represented as 16 bit two's compliment integers (2 bytes). This permits an equivalent decimal number range from +32767 to -32768. This data representation is quite natural to the 6809's machine instruction set which allows CBASIC-3 to produce extremely fast and compact machine code.

Because the compiler supports boolean operations, unsigned 16 bit binary numbers may also be used for many functions. The range for these are: 0 to +65535. These numbers are used for referencing memory addresses in many cases.

CBASIC-3 programs may include numeric constants in either decimal or hexadecimal notation. In the latter case a dollar sign ($) or the characters "&H" must precede the hex value or a pound sign (#) to represent the logical compliment (1's compliment or boolean NOT).

Examples of LEGAL numeric constants:

```
200  -5000  $100  &H1000  -3000  12345  #1  #$5000  $FFF0  &HFFDF
```

Examples of ILLEGAL numbers:

```
9.99 (fraction not allowed except in CIRCLE statement)

100000 (number too large)

+20 (plus sign not allowed, assumed if not minus)
```

Because binary numbers are represented in either unsigned or 2's compliment form, as well as the differences between hex and decimal notation of identical numbers, all the following number constants have the same binary value.

```
-1 $FFFF #0 65635 &HFFFF
```

NUMERIC VARIABLES

Legal numeric variable names in CBASIC-3 consist of a one or two letter name or a single letter and a digit 0-9. Variable names can be longer than two letters if desired but only the first two letters or characters are used for the name. The following are legal variable names:

X N XX ZX R2 A0 ZIP (only ZI is used)

If decalared in a DIM statement, numeric variables may be arrays of one or two dimensions. The maximum subscript size is 32767, therefore the largest one-dimensional array would require 65534 bytes of memory (which is too big to actually be used in Color Computer). Subscripts begin at 0 (BASE 0 subscripting).

When referencing subscripted variables, the subscripts may be numeric constants, variables, or expressions as long as the evaluated results is a positive number from 0 to 32767. CBASIC-3 does not perform run-time subscript error checking for overrange errors which would cost considerably in terms of program size and speed. Two dimensional numeric arrays may be defined and used for a 1 dimensional access which is much faster than a 2 dimension access. If you had the array A(30,100), you could access it as if it was A(3000).

References to two dimensional arrays with less than 255 elements or rows will use the internal 8 by 8 bit multiply instruction for indexing. Numeric arrays with over 255 elements will use a fast 16 by 16 bit multiply to index into the array. Obvioulsy the smaller two dimensional and one dimensional array will have a faster access than a two dimensional array with over 255 elements or rows.

Examples of legal subscription:

N(M) A(1200) Z2(CX) Z4(N,MZ) H(N*(A/B),X+2) R4(N*AZ+K)

CBASIC-3 considers a simple variable with the same name as an array to be the first element of an array. For example: if there is an array A(20,20) using the variable name A without any subscript is equivelent to using A(0,0).

Each numeric variable or element of an array is assigned two bytes of RAM for run-time storage.

ARITHMETIC OPERATORS

The five legal operators for arithmetic are:

```
+  Add
-  Subtract
*  Multiply
/  Divide
-  Negative (UNARY)
```

There are also four boolean operators:

```
& or AND    Logical AND
! or OR     Logical OR
%           Exclusive OR
#           Compliment (UNARY)
```

All of the above operators may be mixed in arithmetic expressions. The boolean operators, operate in a bit-by-bit manner across all 16 bits of the numeric variable.

The order of operation determines in which order CBASIC-3 processes expressions. The compiler will convert arithmetic expressions to an internal form during compilation, and rearrange expressions following the order of operations. In this way CBASIC-3 may produce machine instructions which are shortest and fastest. Expressions are evaluated in the following order:

1. Numeric Functions
2. Unary Negative and Not
3. Multiplication, Division
4. Addition, Subtraction
5. Relational tests <,>,=,<=,>=,<>
6. Boolean operations AND,OR,&,!,%

Parenthesis may be used to alter the normal order of evaluation where required. Some examples of legal expressions:

```
A*B(N,M+4)  $200+ZX  A&B!C*D/F+(H+(J*2)&$FF00)

N+A(Z)/VAL("FOUR")  (C<>D AND A$=B$) OR (C>D AND A$=D$)
```

ARITHMETIC FUNCTIONS

ABS(expr)	The Absolute value of the numeric expression (-324 = 324)
INT(expr)	Converts the numeric expression to an integer (For Color Basic testing)
RND(expr)	Returns a random integer between 1 and the specified expression value (1-32767).
PEEK(expr)	Returns the contents of the memory location determined by the results of the numeric expression.
DPEEK(expr)	Returns the 16 bit value from the two consecutive memory locations determined by the results of the numeric expression.
POS(expr)	Returns the current character position of the specified device number (0=screen, 2=printer, 3=RS-232 port).
POS@	Returns the current PRINT@ location on the screen.
SWAP(expr)	Byte swap of the results of the numeric expression. High order & low order bytes are exchanged.
SGN(expr)	Returns a value indicating whether the expression is positive (+1), negative (-1) or zero (0).
TIMER	Returns the contents or allows setting the timer 0-65535. Ex: TIMER=(expr), Var=TIMER
VARPTR(var)	Returns the absolute memory address for a variable.
OVEREM	Returns the Overflow results of a multiply or the Remainder of a Divide function. Valid immediatly after a multiply or divide only.

ARITHMETIC ERRORS

Arithmetic operations may produce several types of errors which may be detected and processed. Addition and Subtraction may result in a carry or borrow condition. Either one will result in the Carry bit of the MPU's condition code register being set. The ON OVR and ON NOVR statements may be used to detect this condition. This also permits addition and subtraction in larger representation than 16 bits. (See MULTIPLE PRECISION ARITHMETIC)

Multiplication of two 16 bit numbers may result in a product up to four bytes long. CBASIC-3 will detect this error (See ON ERROR GOTO) and preserve the high order 16 bits of the correct 2's compliment result which can be accessed by the OVEREM function.

Division attempted with a divisor of zero will also produce an error which is detected at run-time with the ON ERROR GOTO. The Remainder of a division may be obtained by the OVEREM function: A=OVEREM.

MULTIPLE PRECISION ARITHMETIC

Sometimes it is necessary to deal with numbers larger than the basic 2 byte CBASIC-3 representation. CBASIC-3 allows addition and subtraction of numbers of multiples of 16 bits by means of the ON OVR GOTO an ON NOVR GOTO statements. OVR means overflow (carry or borrow as represented by the MPU C bit) and NOVR means NOT OVERFLOW.

The example below shows addition and subtraction of 32 bit integers using the convention that two variables are used to store each number: A1 and A2 are the first number with A1 being the most significant bytes; and B1 and B2 used similarly. To add A1-A2 to B1-B2 the following subroutine may be used:

```
100 A2=A2+B2 : ON NOVR GOTO 200: ' ADD L.S. BYTES
150 A1=A1+1  : ' ADD 1 TO MS BYTES FOR CARRY
200 A1=A1+B1 : ' ADD MS BYTES
```

To subtract B1-B2 from A1-A2 a similar routine is used:

```
100 A2=A2-B2 : ON OVR GOTO 200 : 'SUB. LS BYTES
150 A1=A1-B1 : RETURN : ' SUB MS BYTES & RETURN
200 GOSUB 150 : A1=A1-1 : RETURN : REM BORROW CASE
```

Extended & Decimal Addition & Subtraction

In many cases it is desirable to use decimal numbers or numbers larger than +/-32767. Although CBASIC-3 cannot handle numbers larger than this directly, simple addition and subtraction of fixed decimal or large numbers can be easily handled using multiple variables. By using multiple variables, each 3 or 4 digits of a large number can be assigned to 1 variable to form a very large number of 6 or more digits. In the following example we will use 2 variables to represent a decimal number with a fixed decimal point for a cents value. The Total value for the sum cannot exceed 32767.99 in this form. This is not the only method in which decimal numbers can be handled, strings can also be used to allow a wider range of decimal values to be input and handled.

In this example ten numbers will be input from the keyboard and added together. The array "V" contains 10 elements each with two variables V(0) and V(1). In this example the numbers input from the keyboard are assumed to have a fixed decimal point for cents and cannot exceed 32767 since they are being input as numeric variables. If a value of 1000 is entered it is assumed to be 10.00, 1222 would be 12.22 and 150 would be 1.50. The maximum input value is thus 327.67 for this example.

```
100 DIM V(1,10): T0=0: T1=0 :' DEFINE ARRAY, CLEAR TOTAL
110 FOR I = 1 TO 10 :' SETUP INPUT LOOP
120 INPUT "ENTER NUMBER TO BE ADDED ";A
130 '
140 ' CONVERT NUMBER TO DOLLARS & CENTS
150 '
160 V(0,I)=A/100: 'ASSIGN DOLLAR VALUE
170 V(1,I)=OVEREM 'ASSIGN CENTS VALUE
180 NEXT I
190 '
200 ' NOW ADD UP THE NUMBERS IN THE ARRAY
210 '
220 FOR I = 1 TO 10
230 T1=V(1,I)+T1 : ' ADD THE CENTS TOGETHER
240 T0=V(0,I)+T0+T1/100 :'ADD THE DOLLARS & CENTS OVER 100
250 T1=OVEREM: ' CENTS = REMANDER OF DIVIDE
260 NEXT I
270 '
280 ' NOW PRINT THE TOTAL FOR THE ARRAY
290 '
300 PRINT "TOTAL = ";
310 '  CONVERT DOLLARS TO STRING A$ WITH $ SIGN
320 A$="$"+STR$(TO)
330 ' CONVERT CENTS TO STRING B$ WITH DECIMAL POINT
340 ' ADD A LEADING ZERO IF ITS VALUE IS LESS THAN 10
350 B$="."+RIGHT$("00"+STR$(T1),2):' ONLY 2 DECIMAL PLACES
360 PRINT A$;B$: GOTO 100 :' PRINT & GO DO IT AGAIN
```

For cases where multiply, divide or even floating point arithmetic must be used, external subroutines may be used. In such cases several compiler features and capabilities may be used to simplify the interface.

1) Use the EXEC or CALL statement to call subroutines.
2) Set up conventions so values are passed to the external subroutines in certain memory addresses that have been assigned CBASIC-3 variable names so the CBASIC-3 program may easily manipulate them.
3) Use CBASIC-3'S string processing capabilities to full advantage in handling I/O and storage of numeric values. Floating point numbers can be passed as ASCII strings.

BUTTON Statement

Syntax: BUTTON(expr)

The BUTTON function is used to tell if a selected Joystick button is pressed. If the selected Joystick button is pressed, the function will return a value of 1. If the button is not pressed a value of 0 is returned. The expression must evaluate to a number between 0 and 3 to be valid. The following values will select the different Joystick buttons:

0 = Right Button 1 (old joystick)
1 = Right Button 2
2 = Left Button 1 (old joystick)
3 = Left Button 2

```
Example:    IF BUTTON(0) = 1 THEN 200
```

JOYSTK Statement

Syntax: JOYSTK(expr)

The JOYSTK function is used to get the horizontal or vertical position of the Left or Right Joystick. It returns a value between 0 and 63 to represent the position. The expression must evaluate to a number between 0 and 3 to be valid. The following values will select the different joystick and their horizontal or vertical value:

0 = Right joystick horizontal coordinate
1 = Right joystick vertical coordinate
2 = Left joystick horizontal coordinate
3 = Left joystick vertical coordinate

```
Example:    H = JOYSTK(0)
```

Run-Time SWITCH variables

```
Syntax:    SWITCHn = numeric expression
           var = SWITCHn
           SWITCH$ = string expression
           var$ = SWITCH$
```

The SWITCH variables are run time variables that occupy the first 16 bytes ($00-$0F) on the direct page of memory used by the CBASIC-3 program. There are eight numeric variable switchs that can be uses or one 16 byte string variable, SWITCH0-SWITCH7 and SWITCH$. They both occupy the same memory space and can be used like any other variables in CBASIC-3. They can be useful for temporary variable storage or for passing variables to & from machine language programs or subroutines and CBASIC-3 programs. Since the SWITCH variables are not initalized by the CBASIC-3 program, they can be useful for passing information to a CBASIC-3 program before it is executed or from one CBASIC-3 program to another CHAINed CBASIC-3 program that uses the same Direct Page of memory.

```
Example:   SWITCH1=A
           AB(I)=SWITCH3
           SWITCH$=A$
           A$="HELLO "+SWITCH$
```

STRING OPERATIONS

CBASIC-3 features a complete set of string processing capabilities which allow CBASIC-3 programs to perform operations on character oriented data. Character type data is represented in CBASIC-3 in "string" form which is defined as variable length sequences of characters terminated with a null (00) character.

String Literals

A string literal or constant consists of a series of characters enclosed in quotation marks:

"This is a string literal"

Any character may be included in a string literal except for the ASCII characters for carriage return or null. A string literal may include up to 255 characters. If a quote is to be included as part of the string two are used so the literal:

"An embedded "" Quote"" = AN embedded " Quote

STRING VARIABLES:

CBASIC-3 allows string variables which may be either single strings or arrays of strings. String variable names consist of one letter and a digit 0-9 or two letters A-Z followed by a dollar sign such as A$, AX$, A1$ or Z$.

String variables may be used with or without explicit declarations. If a string variable is encountered for the first time in the source program as it is being compiled without having been previously declared in a DIM statement, the compiler will assign 32 bytes of storage for the string. The "maximum" number of characters that may be assigned to the variable. If the assignment statement produces a result which has more characters than assigned for the variable the first N characters will be stored where N is the length of the variable storage assigned.

A string variable or array may be declared to have a size of 1 to 255 characters in length if, the string is declared by a DIM statement before it is used (see DIM statement description).

If the string name is declared as an array, the maximum subscript size is 32767. Legal usage of string arrays require that only one subscript (which may be an expression) be used:

A$(5) AX$(x+5) Z1$(A+(N/2)) BB$(X*Y)

String Concatenation

The string concatenation operator + is used to join strings to form a new string or string expressions. For example:

"NEW "+"STRING" produces the new string value: "NEW STRING"

Null Strings

Strings which have no characters are represented as the literal "" which represents an empty string. This is typically the initial value assigned to a string which is to be "built up". The string assignment statement: A$="" is somewhat analogous to the arithmetic assignment A=0 in the sense that both cause a variable to be assigned a defined value of "nothing". This is important because before a string variable is used in a program it has a value which is random and meaningless.

String Functions

CBASIC-3 includes many functions which manipulate strings or convert strings to or from other types. Some of the functions which include $ in their name produce results which are of the string type and may be used in string expressions. In the description of string functions that follow, the notations:

N - refers to a numeric constant, variable or expression.
X$ - refers to a string literal, variable or expression.

The following functions produce STRING results:

CHR$(N)	returns a character which is the value of the number N in ASCII.
LEFT$(X$,N)	returns the N leftmost characters of the string X$. For example the function LEFT$("Example",3) returns "Exa"
MID$(X$,N,M)	returns a string which is that part of the string X$ beginning with its Nth character and extending for M characters. For example: the function MID$("Example",3,4) returns "ampl".
MID$(X$,N,M)=Y	Replace a portion of a string variable "X$" starting at position N for a length of M, with the string Y$.

RIGHT$(X$,N) returns the N rightmost characters of the string X$. An example of this function is RIGHT$("Example",3) returns "ple".

STR$(N) is a function used to convert a numeric value to a string of characters which are decimal digits. For example STR$(1234) returns the "string" "1234". This is the opposit of the VAL function.

STRING$(N,M) is a function which creates a string of N characters in length specified by the ASCII code M. For example: STRING$(10,49) or STRING(10,"1") both produce the string "1111111111", however the numeric form produces almost half as much code as the string from "1".

TRM$(X$) is a function which removes trailing blanks or spaces from a string and is typically used after a string is read from input. For example TRM$("Example ") returns "Example".

HEX$(N) is a function which converts the value of a numeric expression into a string of characters that represent the hexadecimal equivalent of the expression. Example HEX$(255) returns "FF".

INKEY$ is a function that returns a single character string equal to the character value of the key pressed on the keyboard. If no key is pressed on the keyboard, a null string "" is returned.

Note on the preceeding functions: if there are not enough characters in the argument to produce a full result, the characters returned will be those processed until the function "ran out" of input, or a null string, whichever is appropriate. The STR$(N) function will result in a run-time error detectable by the ON ERROR GOTO function if its argument is not legal or convertible to a string.

The following functions have string argument(s) and produce a result which is of numeric type:

ASC(X$) returns a number which is the ASCII value of the first character of the string. For example ASC("Example") returns a value of $45 or decimal 53 which is the ASCII code for the character E. This is the inverse function of CHR$.

LEN(X$) returns the length of a string. LEN("Example") returns a 7. LEN("") returns a value of 0.

INSTR(N,X$,Y$) is a substring search function which searches the string X$ beginning at position N, for the substring Y$. If N is omitted the search begins with the first character in X$. If an identical substring is found the function will return a number which is the position of the first character of the substring in the target string. If the substring is not found the function returns a value of 0. Ex: INSTR("Example","pl") returns a value of 5. INSTR("Example","NO") returns a value of 0. If Y$="" the value of N is returned.

VAL(X$) converts a string of characters for decimal digits and optionally a leading minus sign to a numeric value. This has the inverse effect of STR$. If the string argument is not a legal conversion string (it has too many, non-decimal or not digit characters 0-9) a run-time error detectable by ON ERROR GOTO occurs. For example: VAL("123") returns the numeric value of 123. VAL("THREE") results in an error.

String Operations on the I/O Buffer

Commonly BASICs have limitations because of the input formatting when reading mixed data types. For example; BASIC input conventions cause commas which are part of the input data to break up what are really one long string, etc. CBASIC-3 has a special string variable, BUF$ which is defined to be the contents of the run-time I/O buffer which may be used as any other string variable. BUF$ is 255 bytes in length. Note: the I/O buffer is not used during Random Disk access GET & PUT functions.

The following I/O statement forms are legal for filling or dumping the I/O buffer when used with BUF$:

```
INPUT BUF$              PRINT BUF$
INPUT #N,BUF$           WRITE #N,BUF$
```

Example of using BUF$ as a variable:

```
BUF$=MID$(BUF$+A$,N,M)
```

SWAP String Statement

Syntax: SWAP$(string var,string var)

This command is used to exchange the contents of two string variables without the need for a temporary variable. It is equivalent to something like swapping the variables A$ & B$ which would require code similar to: C$=A$:A$=B$:B$=A$. SWAP$ performs the same operation without having to use an intermediate variable, generates much less code and executes faster. This can be a very handy function and speedup factor when doing string sorts. String literals or functions can not be used, only valid string variables are allowed.

```
Example:  SWAP$(A$,B$)
          SWAP$(A$(I),A$(I+1)
```

String Expressions

String expressions may be created using string variable names, the concatentation operator and string functions. Expressions are evaluated from left to right and the only precedence of operations involved is the evaluation of function arguments is performed before concatenation.

At run-time, string operations are performed on data moved to the "String Buffer", a compiler-allocated area normally 255 bytes long. Because this is always the last data storage area allocated by the compiler (st), any memory available beyond this may be used to allow automatic buffer expansion if operations on extremely complex string expressions are involved.

Examples of legal string expressions:

```
"CAT"
AZ$
LEFT$(BC$,N)
A$+RIGHT$(D1$,XX)+"TH"
MID$(A$+B$,N,LEN(A$)-1)
"AA"+LEFT$(RIGHT$(TRM$(A$)+B$,Z4),X+2)+C$
```

String Comparisons

Strings may be compared in an IF expression the same as numeric expressions. Each character in the string is numerically evaluated by its ASCII character value for relational operations. Remember that puncuation and numeric characters have values that are less than normal text characters. Upper case text characters also have values less than Lower case text characters.

ORG, BASE and DPSET Statements

Syntax: ORG = address
BASE = address
DPSET = address(MSB only)

These statements are used to control how CBASIC-3 assigns memory in and for the object program. The ORG statement is used to assign starting addresses for the object code, and the BASE statement is used to define the addresses used for variable storage. The DPSET statement is used to set the direct page reference value for variable storage. In most cases these statements need not be used at all in standard basic program as the program default values will provide for the optimum program configuration.

Both the ORG and BASE statements may be used as often as desired so memory assignments for variables and data storage may be segmented as desired.

CBASIC-3 uses three internal "pointers" that control how run-time memory is allocated. The "object code pointer" always maintains the address where the next instruction generated by the compiler will be stored. The ORG statement assigns a value to this pointer. When CBASIC-3 is first started up, a default value of $1000 is assigned to the pointer to allow space for the Direct Page ($0F). So unless an ORG statement is processed before the first executable BASIC statement, the programs default starting address is $1000.

For example, the statement:

```
ORG=$4000
```

will cause instructions generated for any following BASIC statements to begin at address $4000. The ORG statement may be used to creat "modules" at different addresses within a single program.

The BASE statement is also used to control memory assignment in a similar manner but it applies to allocation of RAM for variable storage. An internal "data address pointer" is maintained by CBASIC-3 to hold the next address available (at run-time) for variable or temporary storage, in addition to the BASE address pointer. The internal pointer is initialized by the compiler to allocate storage immediatly following the compiled program, and the BASE address pointer is initialized to 0000, which means that it is not being used currently.

CBASIC-3 assigns RAM corresponding to BASIC variables the first time they are encountered in the source program at compilation time. When a "new" variable name is encountered, CBASIC-3 assigns the variable run-time storage corresponding to the current value of the internal data address pointer which is then updated by increasing it by the size of the variable storage assigned, as long as the BASE address pointer is equal to 0000. If the BASE address pointer is not zero, then its value will be used as the next variable storage location and it will be increased accordingly to point to the next available RAM location.

An important function of the BASE statement is to allow specific memory assignments for specific or special variable names. Some of the reasons for this application are as follows:

1. To take advantage of the normally unused upper 32K of ram for large arrays and variable storage.

2. To assign specific variable names and types with memory addresses which have special functions or values. For example addresses of PIA's, X-Pad, 80 coulmn cards, RS-232 cards or other interface devices which have control or status registers, may be given BASIC variable names. A common type of "trick" is to declare the memory used by video displays or graphics memory to be declared as a BASIC array.

3. The BASE statement can assign locations to specific variables without disrupting the normal internal data address pointer, and then allow normal allocation to resume by assigning a value of zero to the BASE pointer (BASE=0000). The BASE statement can also be used for allocating all variable storage by simply setting the location at the beginning of the program and using only the BASE pointer for variable allocation.

When using the ORG and BASE statements the programmer must take care to ensure that there are no conflicts or overlaps between program and data storage, by using assignments which are not overlapped. If the BASE statement is not used, the Compiler will automatically select the correct locations for variable storage.

Sometimes it is useful to declare a variable without generating code at the time it is declared. If the variable is an array, the DIM statement may be used. If it is a simple type, the DIM statement declaration with a size of one may be used for a declaration. For example, to assign the address $FF00 to the variable KB the following sequence may be used.

```
BASE=$FF00
DIM KB(1)
BASE=0000
```

PCLEAR Statement

Syntax: PCLEAR # of Graphics pages

The PCLEAR statement is normally associated with Graphics. In CBASIC-3 the PCLEAR statement is similar to an ORG statement in that it changes the address where the compiled program will be in memory. It will also change the Direct Page reference according to the number of pages to be reserved. The PCLEAR statement must be used in a CBASIC-3 program before any statements that generate machine code, otherwise an error will occur. The number of Graphics pages to clear can be in the range from 0 to 8.

DIM Statement

This statement is used to declare arrays and optionally other simple variables. Arrays must be declared in a DIM statement and may be used to declare more than one array. Arrays may not be redefined in following DIM statements. Array subscripts have a legal range of 0 to 32767.

Numeric Arrays

Numeric arrays may be declared to have one or two dimensions. Two dimensional arrays are stored in row-major order. Each element of a numeric array requires two bytes of storage. A two dimensional array may be accessed as a one dimensional array, this is alowed so large one dimension may be used. Examples of numeric array declaration:

```
DIM B(2000), CX(10,20), D1($10,$20)
```

String Arrays

String arrays may only be one dimensional, however, the DIM statement is also used to specify the string size (1 to 255 characters) so the declaration for a one dimensional string will have two subscripts: the number of strings and the length of each string. A single string may be declared in the DIM statement with a length specification only. Examples:

```
DIM A$(80)          one string of 80 characters
DIM B$(500,72)      500 strings of 72 characters
```

In the example above, A$ is used in the program "Without" any subscripts because it is not an array. B$ would be used in the program with "one" subscript because it is a one-dimensional array. For example:

```
A$=B$(N)
```

Declaring Simple Variables

Because CBASIC-3 allocates memory for variables as they are encountered for the first time, it is often useful to declare a single name so it may be assigned storage at a particular point, but without generating code. This is often the case when it is desired to assign a variable a certain memory address. CBASIC-3 processes a variable declared as an array but used without subscripts in the program as the first element of the array by internally assuming a subscript of (0) for a one dimensional array or (0,0) for a two dimensional array. Because of this a declaration of a variable in a DIM statement with a subscript of 0 is legal, but the variable may be used throughout the program without a subscript.

Example: Suppose a program is to be used to read from and write to a Serial RS 232 interface card at address $FF68 - $FF6B and an X-PAD at address 65376 - 65378 ($FF60-$FF62), and they are to be assigned variable names. A DIM statement at the beginning of the program may be used to assign variable names to these devices:

```
BASE = $FF68       set compiler address pointer
DIM DS(0),CT(0)    declare RS232 data/status/command/ctrl regs.
BASE = $FF60       set address pointer to X-Pad
DIM XY(0),XS(0)    declare x,y reg. and status reg.
BASE = 0000        restore internal data pointer to normal
```

The program may now refer to either the RS-232 port or the X-Pad registers thru the variable names RS, XY, or XS. To access the RS-232 control and command registers:

```
CT=N    to write the command/ctrl regs.
N=DS    to read the data & status regs.
```

or to read the X-Pad x,y coordinates

```
N=XY    read x & Y location regs.
```

REMARK Statement

The REM statement is used to insert comments in the BASIC source program. The first three letters must be REM or the first letter a single quote '. All characters following the REM or quote are considered as comments until an end of line or colon ":" character is encountered. The REM statement does not affect the object program size or speed as it does not generate any code.

TRACE Statement

Syntax: TRACE on/off

The TRACE statement is useful for debugging programs that cause an "FC" Function Call error at run-time. When the compiler is instructed to turn the TRACE mode on, it will automatically generate the code required save the line number of each statement before it is executed. If an error occurs during the execution of the statement and ON ERROR is disabled, the program will pass the line number of the statement in error to Color Basic before the halt is executed. When TRACE mode is enabled it will increase the size of the program by 5 bytes for each line of code. The TRACE mode can be turned ON or OFF at any time within the program. TRACE must be enabled for the ERL function to operate.

Example: TRACE ON

HIRES Statement

The HIRES statement is used to inform the compiler that you would like the Hi-Resolution Text Display functions to be included in your program. The HIRES statement must be used in the beginning of the BASIC program before any program lines that will cause code to be generated. If the HIRES option is included in your program, it will increase the size of it by almost 2K and it will use the Screen memory normally used for the WIDTH 80 display. It will afford you many enhanced screen display formats as well as the ability to use PRINT @ on the 32/40/64/80 column displays. See Appendix D for HI-RES Screen Commands & Functions.

Example: HIRES

MODULE Statement

The MODULE statement is used when you want the compiler to generate the code required to preserve the MPU registers and the Stack of a calling program before initalizing the Stack & Direct Page registers for the compiled programs use. It will also instruct the compiler to ignore the HIRES statement if used and generate the code required to restore the MPU registers and Stack when an END or STOP statement is executed. This can be useful for creating separate modules that can be called from a compiled program. Variable storage will still be allocated normally so variables that are to be passed from the calling compiled program must be coordinated by the BASE and DIM or SWITCH statements if required.

Example: MODULE

CBASIC-3 extensivly changes the I/O structure of the CoCo-3 to add support for the RS-232 port and to improve interrupt handling in 64K modes of operation. Because of these changes, a compiled porgram automatically re-vectors several Color Basic hooks. The program automatically inserts its own vectors in these locations and preserves the old vector information. The program will automatically restore these vectors when the compiled porgram is exited via an END, STOP or CHAIN command. This is important to remember when using more than one compiled program in memory at the same time or using the LOADM & EXEC commands to execute another CBASIC-3 program, since the second or third program will also re-vector these hooks. If the current program was not un-linked before exiting, un-predictable results will occur. The same problem will exist if you try to exit a compiled program into another machine language program or into basic using a CALL or EXEC statement. We have therefore provided two additional commands to allow you to manually Link or Un-link the CBASIC-3 program. These commands are as follows:

UNLINK Statement

This command will un-link or restore the original vector information the same as it was found before the program was executed (normal Color Basic vectors). It would normally be used before you use the CALL or EXEC statement to exit from a CBASIC-3 program. When a program is un-linked, HI-RES, RS-232, ON IRQ and ON ERROR functions will no longer be functional. You can use the UNLINK command at any time within the program, however it is not advisable unless you plan to exit the program.

```
Example:   1020 UNLINK:EXEC $A027: 'unlink & do basic reset
```

CBLINK Statement

This command will allow you to re-link the CBASIC-3 program manually if you have previously un-linked it and executed another program and returned. If the program has not previously been un-linked it will not try to re-link itself, so no conflict will occur.

```
Example:   1020 UNLINK:EXEC $4800: ' go do sort & list
           1030 CBLINK:' restore program links
```

Arithmetic Assignment

Syntax: LET var = expression or var=expression

The expression is evaulated and the result is stored in the variable. Use of the keyword LET is optional.

POKE and Double Byte POKE Assignment

Syntax: POKE address, value
DPOKE address, value

The POKE and DPOKE statements are used to place a single byte (POKE) or double byte (DPOKE) variable or value at a specified location in memory. The address and value can be any valid numeric expression or variable. If numbers are used for both the address and value the shortest and fastest possible code will be generated. When using POKE only the least significant byte of the result is stored. When using DPOKE the full 16 bit value is stored at the address and address+1.

String Assignment

Syntax: LET strvar = strexpr or strvar = strexpr

The string expression is evaluated and the result assigned to the string variable specified. If the result of the evaluation produces a result with a longer length than the size of the result variable, the first N characters only are stored where N is the length of the resulting variable.

DATA Statement

Syntax: DATA value,value,...,value

The DATA statement is used to store information in the program that is to be read in by the program. The data can be either in a numeric or string form, and can be placed anywhere in the program. The compiler will automatically assign it to a data storage area that is invisible to the user. If a DATA statement is used on a multiple statement line, it must be the last statement on the line. All information following the DATA statement up to the end of the line is considered to be valid information.

Examples:
```
DATA 7,Sun,Mon,Tue,Wed,Thur,Fri,Sat
DATA 10,12,14,18,57,99,109,33,Horses,Cows
```

The examples demonstrate that mixed numeric and text can be stored on the same line. It is up to the programmer to know what type of information is stored in the data statements before reading it into the program with the READ statement.

READ Statement

Syntax: READ var, var,...,var

The READ statement is used to read data from a DATA statement as explained in the preceeding paragraphs. The READ statement can be used with more than one variable if desired. When a READ statement is followed by more than one variable, each variable is assigned the next available piece of data. If a READ statement tries to read past the end of all data statements it will automatically be assigned a value of zero for numeric variables and a null string "" for string variables. If ON ERROR handling is enabled it will generate an out of data error.

Example:
```
10 DATA 7,Sun,Mon,Tue,Wed,Thur,Fri,Sat
20 READ N : 'read # of items of data
30 FOR I = 1 TO N
40 READ A$(I): NEXT

10 DATA Sun,Mon,Tue,Wed,Thur,Fri,Sat
20 READ A$,B$,C$,D$,E$,F$,G$,H$
```

In the first example the value of "7" was read from the DATA statement first and then that value was used to count how many items of data were to be READ from the DATA statement. In the second example all the data was read with a single READ statement, only in this case there were 8 variables and only 7 items of data so the variable H$ was assigned a null string value "".

RESTORE Statement

Syntax: RESTORE

The RESTORE statement is provided to allow re-read capability for the DATA statements. When a program is first run, the first READ statement causes the first element of data to be read, each succeeding variable of that READ statement and following READ statements will continue to read the next element of data sequentially. When a RESTORE statement is executed, it causes the "next available data pointer" to be reset to the first DATA statement of the program. The next READ statement executed after a RESTORE will begin reading data from the "first" DATA statement in the program.

Example:
```
05 DIM A(10), B(10)
10 DATA 10,9,8,7,6,5,4,3,2,1
20 FOR I=1 TO 10
30 READ A(I) : NEXT
40 RESTORE
50 FOR X=1 TO 10
60 READ B(X) : NEXT
```

The example shows the array A(10) being assigned the data first, then the RESTORE statement resets the pointer back to the beginning of data again. The array B(10) is then assigned the same values from the DATA statement.

EXEC Statement

Syntax: EXEC address

The EXEC statement is used to directly call a machine language subroutine at the address specified. If the address is omitted it will use the previous EXEC address or the one from the last CLOADM or LOADM. Before jumping to the address, the current Direct Page register contents will be saved on the stack and the Direct Page register will be set to zero for Color Basic ROM call compatibility. Upon returning from the EXECuted program or subroutine, the DP register will be restored from the stack automatically. Obviously if the stack is altered by the EXECuted routine or it does not return with the stack intact, unpredictable results will occur. If you wish to have information from the Executed routine returned to the CBASIC-3 program, use the BASE & DIM or SWITCH variable statements to coordinate returned values.

```
Examples:   EXEC $A282          Execute subroutine at address $A282
            LOADM"TEST":EXEC    Execute subroutine at address 1024
```

CALL Statement

Syntax: CALL address

The CALL statement is similar to the EXEC statement in operation, except that it does not save the DP register or preset it to zero. It requires that the address be specified. It can be useful when you do not want the DP register to be set to zero or if the DP register is set using the GEN statement prior to the CALL statement. The CALL statement translates directly into the machine code for Jump to Subroutine (JSR).

```
Example:    CALL $1000
            CALL $A282
```

FOR/NEXT Statement

Syntax: FOR var = expr TO expr STEP expr
NEXT (var),(var),etc

The FOR/NEXT statement uses a variable "var" as a counter while performing the loop ended by the NEXT statement. If no step value is specified, the increment value will be 1. The FOR/NEXT implementation in CBASIC-3 differs slightly from COLOR BASIC due to a looping method that results in extremely fast execution and minimum length. Note the following characteristics of FOR/NEXT operation:

1. var must be a non-subscripted numeric variable.

2. The loop will be executed at least once reguardless of the terminating value.

3. After termination of the loop, the counter value will be GREATER or LESS than the terminating value depending on the direction of the loop, because the test and increment is at the bottom (NEXT) part of the loop.

4. FOR/NEXT loops may be exited and entered at will.

5. At compile time, up to 16 loops may be active, and all must be properly nested.

6. The initial, step, and terminating values may be positive or negative. The loop will terminate when the counter variable is greater than the terminating value in a forward loop (Ex. 1 to 10), or less than the terminating value in a reverse loop (Ex. 10 to 1).

7. There can be only one NEXT statement for any given FOR loop. Therefore you cannot use the structure: IF A=1 THEN C=C+1 NEXT Y ELSE NEXT Y. This will cause compiler errors and may cause the compiler to loop.

```
Examples:  FOR N = J+1 TO Z/4 STEP X*2
           FOR A = -100 TO -10 STEP -2
           FOR I = 9 TO 3  (REVERSE LOOP)
           FOR I = 3 TO 9  (FORWARD LOOP)

           NEXT X,Y,Z      (more than one loop var)
           NEXT            (end most recent loop activated)
```

GOSUB/RETURN Statements

Syntax: GOSUB line#
RETURN

The GOSUB statement calls a subroutine starting at the line number specified. If no such line exists, an error message will be generated on the second pass. The machine stack is used for return address linkage the same as a normal assembly language program. The RETURN statement terminates the subroutine and returns to the statement following the calling GOSUB. Subroutines may have multiple entry and return points. The GOSUB and RETURN statements compile directly to JSR and RTS machine language instructions, respectively.

IF/THEN Statement

Syntax: IF <expr> <rel.> <expr> AND/OR <expr> <rel.> <expr>
THEN <statement(s)> ELSE <statement(s)>
GOTO <line #> ELSE <statement(s)>
GOSUB <line #> ELSE <statement(s)>

The IF/THEN, IF/GOSUB or IF/THEN/ELSE statements are used to conditionally branch or execute statements, or conditionally call a subroutine based on a comparison of two expressions. Legal relations are:

<	less than
>	greater than
=	equal to
<>	not equal to
<= =<	less than or equal to
>= =>	greater than or equal to

If the statement is an IF/GOSUB the subroutine specified will be called if the relation is true and will return to the next line # following. If an ELSE is used, statements or the line# following it will be executed if the relationship is False. The logical operators AND/OR may be used to test the results of several conditions in one statement.

Examples:
```
IF N = 100 THEN 1210
IF A=1 AND C=2 GOSUB 550
IF XZ=200 OR XY=192 THEN 240 ELSE 1100
IF XZ=200 THEN XY=240 ELSE GOTO 1100
IF A$=B$ THEN C$="YES":D$="NO" ELSE D$=YES"
IF A$>B$ THEN 260 ELSE C$=A$
```

ON ERROR GOTO Statement

Syntax: ON ERROR GOTO
ON ERROR GOTO line#
ON ERR GOTO line#

The ON ERROR/ERR statement provides a run-time error "trap"- the capability to transfer program control when an error occurs.

When an ON ERROR GOTO statement is executed the compiler saves the address of the line number specified in a temporary location. If any detectable error occurs during execution of following statements, the program will transfer control to the line number given in the ON ERROR GOTO statement last executed. This would normally be the line number where an error recovery routine begins.

If the ON ERROR GOTO statement is used WITHOUT a line number specified, it has the effect of "turning off" the error trap - errors in following statements will be ignored.

After an error has been detected, the ERR or ERNO function may be used to access a value which is an error code identifying the type of error which most recently occured. The exact error codes are listed in the appendix. The ERL or ERLIN function may also be used to determine which line number the error occured in, providing that TRACE was ON.

The ON ERROR function if enabled will automatically restore the Direct Page register and initalize the Stack Pointer to the top of the Direct Page (same as default Stack Pointer on startup).

The types of errors that can be detected by ON ERROR GOTO and the types of statements they occur in are listed below:

Divide by zero	Arithmetic expressions
ASCII to Binary conversion error	INPUT, READ, VAL(X$)
Multiply overflow	Arithmetic expressions
Disk, Tape errors	Disk, Tape I/O
Syntax Errors	HI-Res Graphics DRAW, PLAY

Examples of usage:

```
100 ON ERROR GOTO 500
120 INPUT A(N)
130 N=N+1 : IF N=50 THEN 600 :GOTO 120
600 PRINT "ILLEGAL INPUT ERROR - RETYPE": GOTO 120
```

ON BRK GOTO Statement:

Syntax: ON BRK GOTO line#

The ON BRK statement allows you to transfer control to a specified line number when the Break key is pressed. If the Statement is used without a line number it has the effect of turning off Break key detection. If ON BRK is disabled (default) the Break key or Shift @ can be used to pause the display. CBASIC-3 only checks for an ON BRK condition when data is being output to the screen.

Example: ON BRK GOTO 1000

ON-GOTO/ON-GOSUB Statements

Syntax: ON expr GOTO <line#>,...,<line#>
ON expr GOSUB <line#>,...,<line#>

The expression is evaluated and one line number in the list corresponding to the value is selected for a branch or subroutine call. Ex: if <expr> evaluates to 5, the 5th line number is used. If <expr> evaluates to zero or a number greater than the number of lines specified, the statement will be ignored and the next statement on the line or next line will be executed.

Examples: ON A*(B+C) GOTO 200,350,400,110,250
ON N GOSUB 500,510,520,530,100

STOP & END Statements

Syntax: STOP or END

The STOP & END statements are used to terminate execution of a program by causing a Coldstart return to the Color BASIC operating system. If the MODULE statement was used in the program these statements will generate the code required to restore all MPU registers and the Stack Pointer to the program entry conditions.

RUN Statement:

Syntax: RUN

The RUN statement is used to re-execute the compiled program, just as if it were first executed. It will not close any open disk or tape files, like Color Basic. It simply performs a GOTO to the first execution address of the program.

GEN Statement

Syntax: GEN number, number,...., number

The GEN statement allows data or machine language instructions to be directly inserted in the program. The list of values supplied are inserted directly into the object program. If a value given in the list is less than 255, only one byte will be generated for that value reguardless of leading zeros. This function can be very useful for directly inserting machine language subroutines in a BASIC program, as the line # for the beginning of the routine can easily be called via the Basic GOSUB statement and control returned to the calling GOSUB by ending the routine with an RTS ($39) instruction.

Examples: GEN $BD,$A282,$CE,1024 (produces 6 bytes)
GEN 0040,$00,32767 (produces 4 bytes)

CLEAR Statement

The CLEAR statement has no function in CBASIC, it is recognized for conversion of Color Basic programs. It is handled the same as a REMark Statement.

ON RESET GOTO Statement

Syntax: ON RESET GOTO line#

This statement allows a CBASIC-3 program to be re-initialized or continue execution at a specified line# in the program. Normally if the RESET button on the back of the computer is depressed during program execution, the machine is Cold Started and control is returned to Color Basic. The ON RESET statement is typically one of the first statements in a CBASIC-3 program if used, but may be used to re-define the RESET control vector at any time within the program. If an ON RESET statement is executed in the program, the only way to terminate program execution is thru a STOP or END statement. The compiler will automatically generate the proper code to re-initialize the Direct Page and Stack registers and 64K RAM if used.

Example: ON RESET GOTO 5000

STACK Assignment Statement

Syntax: STACK = address

This statement is used to initialize or change the MPU stack pointer register. Normally, the STACK statement is not required in a program as the compiler automatically uses the page of memory immediately prior to the beginning of the program. This is normally adequate for almost all programs, including extensive subroutine nesting and interrupt processing (200 bytes of Stack space). Otherwise, a specific memory area should be dedicated for the stack and the STACK instruction used to set the TOP of the stack (highest address).

Example: STACK = $7FFF (stack builds down)

PAUSE Statement

Syntax: PAUSE ON or OFF

The PAUSE command allows you to select whether or not to allow output to the display to be paused by using the Shift @ key or Break key (CBASIC-3 only). Normally PAUSE is enabled by default when a CBASIC-3 program starts execution so it will work the same as a normal Basic program for stopping a display or detecting an ON BRK condition. However with the addition of ON BRK GOTO and ON KBDIRQ commands in CBASIC-3 the keyboard scan required to detect a pause key being pressed will make the ON KBDIRQ (explained later in the Interrupt commands) not to function properly. If you want to use the ON BRK command, the PAUSE function must be on since CBASIC-3 only checks for the Break key when data is being output to the Screen.

Example: PAUSE OFF

SIGN Statement

Syntax: SIGN ON or OFF

The SIGN command allows you to select whether or not to add a leading space to positive numeric values output to a device. Normally CBASIC-3 supresses this leading space so that multiple numeric variables can be output together to represent larger numbers. Since Color Basic normally outputs this leading space, many programs expect it to be there when doing number to string conversions, etc. To make CBASIC-3 output the extra space, use the SIGN ON command.

Example: SIGN ON ‹enable leading sign space›

Interrupt processing is not easily understood unless you are familar with the hardware of the machine and machine language programming. They can easily hang up a program or cause the system to crash unless used carefully. We have tried to make them easy for you to use by doing most of the tedious processing required for interrupt handling, but if not properly understood you can still have a lot of difficulty using them, so please beware. Also note that we recommend that only simple commands be used within an Interrupt processing subroutine, do not attempt to use any I/O commands or string manulipation commands since you can not determine what other functions may have been in progress when the Interrupt condition was detected and you may make the results of the function that was in progress totally invalid or even hang the system.

ON INTERRUPT Statements *** Not available in Basic

```
Syntax:  ON KBDIRQ GOTO line#   (Keyboard interrupt)
         ON TMRIRQ GOTO line#   (12 bit Timer interrupt)
         ON SERIRQ GOTO line#   (Serial data interrupt)
         ON IRQ    GOTO line#   (60 cycle/other interrupt)
```

The ON Interrupt commands allow you to do real time processing based on interrupt conditions. The Keyboard, Timer and Serial data interrupts are not normally enabled (or available in Basic) and must be enabled via the "IRQ" statements after each time the interrupt occurs. If enabled, and one of these interrupts occur, the detected interrupt type will be disabled from re-occuring until an "IRQ=" statement is used to re-enable them. The reason for this automatic disable feature is that an interrupt may be processed continuiously in error. For instance if a Keyboard interrupt is detected and processed, the Return from Interrupt is executed and the Key is still pressed on the keyboard (Guaranteed). Which means that the Keyboard interrupt would be processed possibly thousands of times for a single key stroke.

A Keyboard interrupt can be generated by any key on the keyboard if the data line from the keyboard PIA ($FF02) output is at a zero level for that key column. For example to enable all keys for interrupt detection you would poke a 00 value at $FF02, or to enable the key column with Enter,@,H,P,X,0 and 8 keys you would poke a 01 at $FF02. Another note is that any time an INKEY, INPUT or other command that causes a keyboard scan (PRINT with PAUSE enabled) will change the value of $FF02. A good way to process a Keyboard interrupt is to simply set a flag variable and let the Main program do the acutal Key scan with an INKEY and then re-enable the KBDIRQ when a key is no longer pressed. The KBDIRQ function should be disabled when using normal INKEY, INPUT and GETCHAR commands from the keyboard by using the "IRQ=" statement before attempting keyboard input.

A Serial data interrupt is generated when the RS-232 input data line on the computer goes from a zero state to a one state (serial data bit = 1 or printer status goes to not ready). It can not be used to detect a start bit for serial data since it is a one to zero transition which makes the Serial data interrupt of little value for Serial communications. However, it may become more useful in an future revision of the Coco3 if it becomes programmable by changing the inverter gate used to an Exclusive or gate with one input tied to one of the pia output lines (hint to R.S.). Until then it works basically the same as the KBDIRQ in that once detected it is disabled until re-enabled by use of the "IRQ=" statement.

A Timer interrupt is generated by the 12 bit programmable timer built into the GIMI chip (in case you didn't know). The Timer register at address $FF94 & $FF95 is loaded with a value least significant byte first ($FF95), with the count automatically beginning when the most significant byte ($FF94) is loaded. As the count falls thru zero, an interrupt is generated (if enabled), and the count is automatically reloaded. As with the Keyboard & Serial interrupts, the Timer interrupt is disabled until re-enabled by the "IRQ=" statement. You can select the input clock to be either 63 micro seconds or 70 nano seconds by the TINS input (bit 5 of $FF91). Default is the 70 nsec clock and we do not recommend that you fool with it since that register also controls the Memory Managment Unit Task Register Select, which if changed at the wrong time can crash the system instantly and it is not a readable register (so you never can tell whether the TR bit or TINS bit is On or Off).

The normal IRQ interrupt is generated every 1/60th of a second by the vertical retrace interrupt in the computer (the same as the Coco 1 & 2), and is used for the TIMER value increment as well as Sound and Play commands for timing. The ON IRQ statement will be executed if any IRQ interrupt is generated including KBD, Serial or Timer if a handler is not set up for that particular interrupt by an ON TMRIRQ/KBDIRQ/SERIRQ statement. Essentially it is a catch all interrupt handler. The 1/60th second interrupt is never disabled automatically like the other interrupts, so it will occur continuously unless disabled by some other means. Since this is a normal interrupt function, CBASIC-3 will automatically handle the interrupt even if you do not have and ON IRQ handler setup, so don't think you have to have one in a CBASIC-3 program, you don't.

A few points to remember, ALL interrupt handling subroutines must end with a RETI statement or you will get a crashed system. If you wish to disable one of the interrupt handlers that have been in use, Use the same statement without a line# (Ex. ON KBDIRQ GOTO) instead of (ON KBDIRQ GOTO 1000).

IRQ Statements

Syntax: IRQ = value
IRQ ON/OFF

The IRQ statements are used to enable or disable IRQ & FIRQ interrupt detection either entirely or partially. The IRQ ON statement is used to disable the detection of all IRQ interrupts by setting the 6809 MPU mask bits for interrupt detection. The IRQ OFF statement clears the 6809 MPU mask bits and allows the detection of all FIRQ & IRQ types. It is recommended that you use a IRQ ON command before setting up ON Interrupt handlers and then using the IRQ OFF statement to enable them when finished.

GIME

The "IRQ=" statement is used to selectively enable or disable interrupt conditions. There are six different interrupt conditions that can be enabled by this statement which gives 64 possible interrupt combinations. They are selected by adding together the bit values of the interrupt enable bits. To activate an interrupt condition, you set the bit on and off to de-activate it.

1 =	Cartridge IRQ	2 =	Keyboard IRQ
4 =	Serial data IRQ	8 =	Vertical Border IRQ
16 =	Horizontal Border IRQ	32 =	Interval Timer IRQ

These values are OR'd together and stored into $FF92 IRQ or $FF93 FIRQ

If you wanted to enable the Keyboard and Timer interrupts you would use a value of 34 (2 for the KBD plus 32 for the Timer). If you are working with more than one interrupt, you should keep a variable with the value of all interrupt conditions and use bit operators like AND (&) and OR (!) to set and reset the bits to be enabled.

Interrupt Handler Example

```
10 IRQ ON : ON TMRIRQ GOTO 100 : IRQ = 32
20 POKE $FF95,0 : POKE $FF94,4 : IRQ OFF
30 TI = 0 : ' COUNT= 0, IRQ EVERY 1024 CLOCKS
40 PRINT @ 0, "TIMER COUNT = ";TI
50 GOTO 40
100 TI = TI + 1 : ' ADD 1 FOR EACH TIMER IRQ
110 IRQ = 32 :     ' RE-ENABLE TIMER IRQ
120 RETI
```

Unnecessary

These are GIME interrupts and require $FF90 to be set up to allow interrupts to reach the CPU.

Example: $FC for COCO2 or $7C for COCO 3 screen modes

Other ON INTERRUPT Statements

Syntax: ON FIRQ GOTO line# Must save registers manually FIRQ
ON NMI GOTO line# GEN $3406 to PSHS D at start of IRQ code
ON SWI GOTO line# GEN $3506 to PULS D before RETI

These statements are used for generating programs where interrupts are processed by specific service routines rather than by the normal Color Basic service routines. When encountered in a program these statements cause the absolute address of the Basic program line specified to be stored at the interrupt vector addresses in the operating system memory. The line number specified should be the beginning of the interrupt service routine which would typically service the device causing the interrupt. This routine is similar to a BASIC subroutine except it is terminated by an RETI (return from interrupt) statement instead of a RETURN statement. These are not normally used unless you have a good understanding of how the M6809 interrupt structure works.

INTERRUPT RETURN Statement

Syntax: RETI

The RETI statement is used to terminate an interrupt-caused routine by loading the MPU register contents prior to the interrupt from the machine stack, and resuming program execution from the point where the interrupt was acknowledged. This statement corresponds directly to the machine language RTI instruction.

Interrupt Simulation Statements

Syntax: IRQ : NMI : FIRQ : SWI

These commands allow you to simulate an interrupt via software in a CBASIC-3 program. They can be useful for testing interrupt handling routines without having to use live interrupts and for special function handling in a program. These commands cause the current processor registers to be saved on the stack & interrupt masks to be set the same as their hardware counterparts. On interrupt handlers should use the RETI command to exit the routine the same as if it were handling a hardware interrupt. All interrupt simulation commands generate 11 bytes of code to simulate the interrupt except the SWI command which generates only 1 byte for the SWI code. Note that SWI2 & SWI3 interrupt vectors are reserved for use by the compiled programs use and are not available for use by the programmer.

LPOKE & DLPOKE Statements

```
Syntax:  LPOKE page#,offset,value     *** Different from Basic
         DLPOKE page#,offset,value *** Not available in Basic
```

The DPOKE and LPOKE commands are used to place a single byte (LPOKE) or double byte (DLPOKE) variable or value in a specified extended memory location (00000- 7FFFF). The page# value is used to select which 64K bank (0-7) is to be used. The offset selects which address in the selected page to use (0-FFFF) and the value is the data to be stored at that location. The page, offset and value can be either numeric or variables used to specify the information. When using the LPOKE statement only the least significant byte of the result is stored and DLPOKE will store the full 16 bit value.

```
Example:  DLPOKE 6,0,255
          LPOKE 6,0,&HFFFF
          LPOKE P,OF,VA
```

LPEEK & DLPEEK Statements

```
Syntax:  A=LPEEK(page#,offset)     *** Different from Basic
         A=DLPEEK(page#,offset)  *** Not available in Basic
```

The LPEEK & DLPEEK commands are used to examine or get the information stored in a specified Extended memory location. The page# specifies which 64K bank of memory (0-7) and the offset selects which address within that 64K block is to be accessed (0-FFFF), the same as the LPOKE command. If the LPEEK command is used a single byte value will be read and stored in the least significant byte (0-255 only) and the DLPEEK command will return a full 16 bit value from the two consecutive bytes.

```
Example:  A = LPEEK(6,10)
          A = DLPEEK(P,OF)
```

RAM64K Statement *** Not available in Basic

Syntax: RAM64K page#

The RAM64K statement tells the compiler that a full 64K of RAM is to be made available in the computer for variable storage etc. The compiler will automatically generate code to allow access to the upper 32K of ram during program execution. This normally unused 32K of memory can be used for any variable or array storage except for Disk related file buffers and Fielded variables. It is especially handy for large Arrays and string variable storage. This area of memory begins at address $8000 and extends up to $FDFF, a total of 32,255 bytes of extra memory storage. To define variables in this area, it is best to use the BASE and DIM statements.

```
Example:  BASE=$8000
          DIM A1$(200,80),A2$(50,255),AZ(1600)
          BASE=0000
```

The preceeding example demonstrates how easy it is to assign variables to the upper 32K of RAM, the two string arrays A1$ and A2$ occupy 28,750 bytes and the numeric array AZ occupys 3200 bytes of RAM. The BASE pointer is then restored to zero to allow any further variables to be assigned address space immediately following the program.

The RAM64K statement for CBASIC-3 allows you to select any 32K bank of memory to be used in place of the upper 32K of memory where Basic normally resides. In the CoCo-3 you are normally in the ALL RAM mode and a modified image of the Basic ROM's is stored there and used for I/O calls and some other functions in a CBASIC-3 program. You can still use the upper portion of memory $8000-$FDFF for variable storage etc. but with a little twist. You must tell CBASIC-3 what the starting page# of the 32K bank of memory you want to use in the upper 32K area to replace the Basic ROM' code. This means that you can select any 32K block of ram available in the machine to be access as the upper 32K, which gives you about 420K of storage space if desired. The page# specified can be a number or variable in the range of 0-59 to select the starting 8K page (60-63 is the normal 64K being used) . For example if you wanted to select the Extended Hi-Res Graphics pages (320/640 * 192) which reside in memory from $60000-$67FFF (32K total) you would use a value of 48 decimal or $30 hex to start at $60000 (8 blocks of 8K for each 64K). If you want to deselect the upper 32K of memory to the normal ROM image use a value of ~~255~~. To return to normal Basic ROM map, you must use RAM64K 60

```
Example:  RAM64K 48
          RAM64K $30
          RAM64K 255 This does nothing
```

RAM ON/OFF Statements

Syntax: RAM <ON/OFF>

The RAM statement allows manual control of the upper 32K of memory space address mode. The RAM ON statement, switches the Basic ROM's off and enables access to the upper 32K of RAM (normal CoCo-3 mode) which normally contains a modified image of the Basic ROM's. The RAM OFF statement does just the opposit, it disables the upper 32K of RAM and enables the Basic ROM's to occupy the upper 32K of address space. These two statements can be useful when RAM64K is not being used and access to some part of the Basic ROM's is needed, you simply enable the ROM's with a RAM OFF statement and when finished, restore to the RAM64K mode by using a RAM ON statement. These statements can be used whether or not the RAM64K statement has been used to allow accessing these areas of memory.

When using the RAM ON/OFF option, it is necessary to either mask interrupts with the IRQ ON statement or provide ON IRQ and ON FIRQ interrupt handling.

LPCOPY Statement *** Not available in Basic

Syntax: LPCOPY source TO destination

The LPCOPY statement is used to copy the contents of any 8K page (0-63) of memory to any other 8K page (0-63). The "source" and "destination" are numeric constants or expressions between 0 and 63 specifying memory pages. This can be very handy for swaping info to and from the Extended Graphics screens which are normally not accessable.

Examples: LPCOPY 1 to 48
LPCOPY AX to AY

The first example would copy the 8K block on page 1 (02000-03FFF) to page 48 (60000-61FFF). The second example demonstrates the use of variables to specify the source and destination pages.

WIDTH Statement

Syntax: WIDTH value

The WIDTH command sets the text screen resolution to either 32 (32 * 16), 40 (40 * 24) or 80 (80 * 24).

Example: WIDTH 80

LOCATE Statement

Syntax: LOCATE (x,y)

The LOCATE command allows you to position the cursor to any column (x) and line (y) position on the 40 or 80 column text screens. When used on a WIDTH 40 screen, the x position can be 0 to 39. When used on a WIDTH 80 screen, the x position can be 0 to 79. On either screen the y position can be 0 to 23.

Example: LOCATE (3,10)
LOCATE (X,Y)

ATTRIbutes Statement

Syntax: ATTR foreground, background,Blink,Underline

The ATTRibutes command allows you to select the foreground (Character) and background colors for the WIDTH 40 and 80 text display modes. These can be in the range of 0-15 to select a pallette color. Optionally you can select if the characters are to be Blinking and/or Underlined by following the fore/background colors with the letter "B" for Blinking or "U" for Underlining. Attributes stay in effect until the next ATTR command is executed.

Example: ATTR 3,2,U (select underline on)
ATTR F,B,B (select blink on)

HSTSTus Statement

Syntax: HSTAT v1,v2,v3,v4

The HSTAT command is used to get information about the 40 or 80 column text screen cursor position. The values returned in the variables are: v1=character code, v2=character attribute, v3=cursor x coordinate and v4=cursor y coordinate.

Example: HSTAT A,B,C,D

In the description of the following Low Resolution Graphics Statements the notations:

c refers to a numeric constant or expression in the range of 0 to 8 and represents a specified Color for the Low Resolution Text Display.

x refers to a numeric constant or expression in the range of 0 to 63 and represents the X coordinant (horizontal position) on the Low Resolution Text Display.

y refers to a numeric constant or expressing in the range of 0 to 31 and represents the Y coordinant (vertical position) on the Low Resolution text Display.

CLS Statement

Syntax: CLS(c)

The CLS statement is used to clear the Low Resolution Display Screen to a specified color "c". If a color is not specified, Green is used by default. If the HIRES statement has been used to include the Hi-Resolution Display package, a CLS statement without a color will cause the Hi-Res Text Screen to be cleared.

```
Example:  CLS(2)
          CLS
          CLS(N)
```

SET Statement

Syntax: SET(x,y,c)

The SET statement is used to set a graphics dot at a specified Text Screen location to a specified color. The x,y coordinants can range from 0 to 63 and 0 to 31 respectively, and the color specified can range from 0 to 8. Any one or all the arguments can be a constant or variable expression.

```
Example:  SET(14,13,3)
          SET(x,y,4)
```

RESET Statement

Syntax: RESET(x,y)

The RESET statement is just the opposite of the SET statement. It is used to reset or clear a point on the Low Resolution Text Screen. The x,y coordinants can be a constant or variable the same as the SET statement.

Example: RESET(14,4)
RESET(X,Y)

POINT Statement

Syntax: POINT(x,y)

The POINT statement is used to test whether a specified Graphics cell on the Text Display is on or off. The x,y coordinants can be a constant or variable expression the same as the SET & RESET statements. The value returned is -1 if the cell is in a Text Character mode, 0 is returned if it is off, and the color code 1-8 is returned if it is on.

Example: A=POINT(14,4)
A=POINT(X,Y)

SOUND Statement

Syntax: SOUND tone, duration

The SOUND statement allows you to generate a sound thru the TV speaker with a specified tone for a specified duration of time. The tone and duration can be either constants or variable expressions in the range of 1 to 255.

Example: SOUND 128,3
SOUND T,D

The Medium Resoulution Graphics statements in CBASIC-3 are almost identical to those in Extended Color Basic. Some brief descriptions of the statements are given to show differences and examples of their usage. For more information on these statements and graphics refer to the Extended Color Basic Manual.

In the description of the following Medium Resolution Graphics Statements the notations:

x specifies the X-coordinant (horizontal position) on the graphics display area and is a numeric constant or expressing from 0 to 255.

y specifies the Y-coordinant (vertical position) on the graphics display area and is a numeric constant or expressing from 0 to 191.

c specifies an available color code and is a numeric constant or expression from 0 to 8. This is optional in many statements; if omitted, the foreground color is used.

PMODE Statement

Syntax: PMODE N, page

The PMODE statement sets the graphics resolution and optionally the memory page to start on. The PMODE value ranges from 0 to 4 with 4 being the highest resolution mode (256*192). The starting page "page" is a numeric expression or constant from 1 to 8, and specifies which 1.5K memory page you wish to start on. This is optional; if omitted, the previously set page is used. If the PMODE statement is never used, the computer defaults to PMODE 2,1. For more information see the Extended Color Basic Manual.

```
Examples:  PMODE 4,1
           PMODE 3,P
           PMODE 4
```

The first example sets the graphics mode to 4 starting on the 1st page of graphics memory. The second example selects mode 3 which is 128*192 in four colors and the starting page is specified by the variable "P". The third example simply sets the mode to 4 without a starting page.

COLOR Statement

Syntax: COLOR foreground, background

The COLOR statement allows you to change the graphics foreground and background colors (within the available choices). The "foreground" and "background" colors are numeric constants or variable expressions from 0 to 8, and represent the color codes.

```
Examples: COLOR 5,7
          COLOR 7,5
          COLOR FG,BR
```

The first two examples simply show constants being used for the foreground and background colors. The second example reverses the foreground and background colors from the 1st example. The third example shows variables being used for the color codes.

SCREEN Statement

Syntax: SCREEN type, color

The SCREEN statement is used to select between the Text or Graphics screen "type", and to optionally select a color set "color". The "type" is either an 0 for text screen or a 1 for graphics screen. The "color" set is either an 0 or a 1 to select which color set is to be used.

```
Examples: SCREEN 1,1
          SCREEN 1,0
          SCREEN 0,1
```

PSET Statement

Syntax: PSET(x,y,c)

The PSET statement is used to set a single point on the graphics screen to a specified color. The x and y coordinants are used to specify exactly which position on the screen you want to set. The c argument is used to specify the color the dot on the screen will have.

```
Examples: PSET(0,0,8)
          PSET(128,96,8)
          PSET(X1,Y1,8)
```

The first example will set a dot in the top left corner of the screen and the 2nd example sets a dot in the center. The third example uses variables for the x,y coordinants.

PRESET Statement

Syntax: PRESET(x,y)

The PRESET statement does the exact opposite of the PSET statement. It "resets" a dot in the screen to the background color. The x and y arguments are used to specify exactly which dot on the screen is to be reset. Notice that you don't have to specify the color with PRESET since the computer automatically uses the background color.

```
Examples: PRESET(128,96)
          PRESET(X1,Y1)
```

The first example will reset the dot at the center of the screen and the second example demonstrates the use of variables for the coordinants.

PPOINT Statement

Syntax: PPOINT(x,y)

The PPOINT statement is similar in form to the PRESET statement, but instead of reseting the specified dot on the screen, it tests the color of a specified graphics point. Your program may then use the information any way you choose. The PPOINT statement returns a value from 0 to 8 to represent the color of the specified graphics point.

```
Examples: C=PPOINT(128,96)
          IF PPOINT(X1,Y1) = 8 THEN 500
```

The first example will get the value of the color from the point in the center of the screen and assign that value to the variable C. The second example demonstrates the use of PPOINT in an IF/THEN statement, that is testing to see if the point at location X1,Y1 is orange in color, if so it will transfer control to line 500.

PCLS Statement

Syntax: PCLS color

The PCLS Statement is used to clear the graphics screen to a specified color 0-8. If a color is not specified, the screen will be cleared to the background color. This serves the same function for Hi-Res graphics as CLS does for the text screen.

Examples: PCLS
PCLS 6

The first example would simply clear the screen to the background color. The second example would clear the screen with the color "cyan" (color code 6).

LINE Statement

Syntax: LINE(x1,y1)-(x2,y2),a,b

The LINE statement is used to draw a line, box or rectangle on the graphics screen. The x1,y1 coordinants are used to specify the starting point on the screen and the x2,y2 coordinants are used to specify ending point for the line. The line is then drawn by the computer between these two points. The "a" argument is used to tell the computer whether to draw the line using the pre-specified foreground color (PSET), or to use the pre-specified background color (PRESET). The PRESET function may be compared to "erasing" rather than drawing on the screen, since the background color makes the line invisable.

The "b" argument is an option that allows you to draw a "Box" or rectangle without having to draw four separate lines. All you have to do is specify two of the opposing corners for the square in x1,y1 and x2,y2, and add ",B" to the statement. You also have the option to add an "F" to the optional argument ",B" to produce ",BF". This will let you "fill" the box with the foreground color to produce a solid box.

Examples: LINE(0,0)-(255,191),PSET
LINE(64,48)-(96,64),PSET,BF

The first example will draw a line from the top left corner of the screen to the bottom right corner. The second example will draw a rectangle 32 points across and 16 points down in the middle of the screen and fill the box with the foreground color.

PCOPY Statement

Syntax: PCOPY source TO destination

The PCOPY statement is used to copy the graphics content of one memory page to another. The "source" and "destination" are numeric constants or expressions between 1 and 8 specifying memory pages.

Examples: PCOPY 3 to 8
PCOPY AX to AY

The first example would copy the graphics on page 3 to page 8. The second example demonstrates the use of variables to specify the source and destination pages.

PAINT Statement

Syntax: PAINT(x,y), color, border color

The PAINT statement allows you to "paint" any shape with any available color. The x,y coordinants are used to specify where on the graphics screen the painting is to begin. The "color" parameter specifies the color code of the paint 0-8. The "border color" parameter tells the computer the color code of the border at which the painting is to stop. If the computer reaches a border other than the specified color, it will paint over that border.

Examples:
```
10 PMODE 3,1
20 PCLS
30 SCREEN 1,1
40 CIRCLE(128,96),90
50 PAINT(128,96),8,8
60 GOTO 60
```

The sample program will draw a circle in the center of the screen and paint in orange.

CIRCLE Statement

Syntax: CIRCLE(x,y),r,[color],[hw ratio],[start],[end]

The CIRCLE statement will allow you to create a full circle, a partial circle or an ellipse using a single Basic statement. The only arguments required to make a circle are the center point coordinants (x,y) and a radius "r", all other arguments are optional. The radius "r" specifies the circle's radius in units from 0 to 255, each unit of measurement is equal to one point on the screen. The optional "color" specifies an available color 0-8, default is the foreground color. The height/width ratio "hw" is optional, it specifies the ratio or the circle's "width" to it's "heigth", if not specified, a value of 256 is used (1:1). A value less than 256 results in a circle "wider" than it is high, a value over 256 results in a circle "Higher" than it is wide. The start & end options allow you to draw just part of a circle (an arc). To use this option, specify the point where the arc is to begin (0-255), insert a comma, and then the point where it is to end (0-255). The starting point (0) for any circle is equivalent to 3 o'clock on a clock, 64 would be 6 o'clock, 128 would be 9 o'clock and 192 would be 12 o'clock. To use the start and end options, you must specify the "hw" ratio, for a normal arc, use hw=256. For more information on the "CIRCLE" statement refer to the Extended Color Basic Manual.

```
Examples: CIRCLE(128,92),95
          CIRCLE(128,92),30,1,256,64,192
          CIRCLE(X1,Y1),30,1,HW,ST,EN
```

The first example demonstrates a simple circle drawn at the center of the screen. The second example demonstrates the use of all options to draw a half circle from 6 o'clock (64) to 12 o'clock (192). The last example is similar except variables are used instead of constants.

Please note that the "hw", "start" and "end" arguments in CBASIC-3 differ from those in Color Basic since they are fractional numbers. If these items are specified as constants in the CIRCLE statement the normal Color Basic decimal format will be accepted by the compiler. When variables are used the values assigned to the variables must conform to the specifications listed above for "hw", "start" and "end".

DRAW Statement

Syntax: DRAW string expression

The DRAW statement is used to draw a line or series of lines, by specifying its direction, angle, and color. The string expression may be a "literal", string variable or expression used to contain the DRAW statement commands. The DRAW commands are as follows:

Motion cmds.	M = Move the draw position
	U = Up
	D = Down
	L = Left
	R = Right
	E = 45 degree angle
	F = 135 degree angle
	G = 225 degree angle
	H = 315 degree angle
	X = Execute a substring & return
Mode cmds.	C = Color
	A = Angle
	S = Scale
Option cmds.	N = No update of draw position
	B = Blank (no draw, just move)

The Motion commands tell the computer where to start drawing on the screen (Mx,y), which direction to draw in (U,D,L,R,E,F,G,H), and how many dots to draw (U25,D25,E30,.. etc.). The motion command Mx,y; positions the cursor to a specified x,y point on the screen, to avoid unwanted lines on the screen preface the M command with the letter B (BM 128,96). The M command can also specify a position "relative" to the current x,y position by preceeding each of the coordinants with a "+" or "-" sign (BM+15,-15).

The Mode command "Cx" allows you to specify a color code 0-8 to be used while drawing (C7).

The Mode command "Sx" allows you to "scale" a drawing up or down, where x is a number from 1 to 62 to indicate the scaling factor in 1/4 units. A scale of 4 = full scale 4/4, a scale of 1 = 1/4 scale, a scale of 8 = double scale 8/4 and so on up to 62. After an Sx command all motion commands will be scaled accordingly until the next Sx command.

The Mode command "Ax" allows you to specify the angle at which a line is to be drawn, 0 = 0 degrees, 1 = 90 degrees, 2 = 180 degrees, and 3 = 270 degrees. All lines drawn following an Ax command will be drawn relative to the angle displacement specified by Ax.

The option "B" blank, has already been mentioned in relation to the Move command. It can also be used to preceed any motion command to cause a blank line to be drawn. This only affects the line immediately following the "B" blank option.

The option "N" can be used to tell the computer "not" to update the x,y location after drawing a line, but to return to the current x,y location before doing the next command. This only affects the command immediately following it.

The last Motion command "X" allows you to execute another DRAW string assigned to the string variable immediately following the command (XA$). The computer will execute this DRAW string and then return to the next command following.

```
Examples:  DRAW "BM128,96;E25;F25;G25;H25"
           DRAW A$
```

The first example moves the draw position to the center of the screen 128,96 and draws a box. The second example shows the use of a string variable for the DRAW string. For more information and examples of using the DRAW statement refer to the Extended Color Basic Manual chapter 7.

GET & PUT Statements

Syntax: GET (x1,y1)-(x2,y2), destination, G
PUT (x1,y1)-(x2,y2), source, option

The GET and PUT statements are used to "get" a rectangular area which contains a graphics display, store it in an array, then "put" the array back on the screen at a later time. The x1,y1 and x2,y2 coordinants are used to tell the computer where the upper left corner and lower right corner of the rectangular graphics area is located on the screen to GET or PUT. The "destination" for GET is the name of a pre-defined numeric array that will be used to store the rectangle's contents. The "G" parameter for GET is optional, but if used specifies that the rectangle will be stored in the array with "Full GRAPHIC" detail.

The "source" parameter for the PUT statement is the name of a pre-defined numeric array that contains the data or previously stored GET rectangle, that is to be written to the display. The "options" for the PUT statement determine how the data is to be written to the display. They consist of the following:

PSET Set each point that is set in the source array.

PRESET Reset each point that is set in the source array.

AND Logically AND each point in the array with each corresponding point in the destination rectangle. If both points are set then the screen point will be set, otherwise reset.

OR Logically OR each point in the array with each corresponding point in the destination rectangle. If either point is set then the screen point will be set.

NOT Reverses the state of each point in the destination rectangle reguardless of the PUT arrays contents.

Before using the GET or PUT statements, an array must be defined to store the graphics data. The size of the array is determined by the size of the display rectangle. It must be large enough to hold all the data, but not too large, or memory space will be wasted. Since CBASIC uses 2 byte integer variables for storage, it is easy to determine how large an array is required to hold the data. First, you must obtain the length and width of the rectangle by subtracting x2 from x1 and y2 from y1. Divide the X value by 16 rounding up to the next higher even number and add 2, then multiply that by the Y value. Now you have the number of elements for the array. You can use either a one or two dimensional array.

For example: if a graphics rectangle is 40 by 20, you have 40/16=2.5 rounded up equals 4. Multiplied by 20 for the Y value gives a total of 80 elements. This is the number of elements for the array DIM X(50). If we had a large rectangle 180 by 125, this gives us 180/16=12 (rounded up) and 12 * 125 equals1500 elements. You could use a (12,125) array or a one dimensional array of 1500. There are several other ways to figure out the dimension size, this is just a simple straight forward way that seems to always work.

One more note, if you use the "G" option in a GET statement, you must use one of the options available for PUT or "garbage" will appear when you put the rectangle back on the screen. For more information on GET and PUT see the Extended Color Basic Manual chapter 8.

Examples:
```
10 DIM X(80)
20 GET (10,10)-(30,30),X,G
30 PCLS
40 PUT (100,100)-(120,120),X,PSET
50 GOTO 50
```

The sample program simply "gets" the 20 by 20 rectangle from the screen and stores it in array X. It then clears the screen and "puts" the rectangle at a different location on the screen.

PLAY Statement

Syntax: PLAY string expression

The PLAY statement allows you to play music thru the speaker in the TV set. It allows you to control the notes, octave, volume, note length, tempo, pauses, sharps, flats and allows execution of substrings. These functions are controled with the following commands.

Notes	A letter from "A" thru "G" with the a "+" or "#" to denote a sharp note and "-" to denote a flat note. Optionally the numbers 1 thru 12 can be used to represent the notes: C, C#/D-, D/E-, E-/D#, E/F-, F/E#, F#/G-, G, G#/A-, A, A#/B-, B respectively.
Octave	The letter "O" followed by a numeral from 1 to 5 is used to represent the octave. If omitted, octave 2 is used. The higher the value the higher to notes will be.
Note-length	The letter "L" followed by a numeral from 1 to 255 is used to represent the note length. If omitted, the current length is used. The value represents the length of the note as follows: 1=whole note, 2=1/2 note, 4=1/4 note, etc..
Tempo	The letter "T" followed by a numeral from 1 to 255 is used to represent the tempo. If opmitted, T2 is used. The higher the value the faster the tempo will be.
Volume	The letter "V" followed by a numeral from 1 to 31 is used to represent the Volume. If omitted, V15 is used. The higher the volume will be.
pause-length	The letter "P" followed by a numeral from 1 to 255 is used to represent a pause. The values represent the length of the pause, 1=whole note, 2=1/2 note, 4=1/4 note, etc..
Execute string	The letter "X" followed by the name of the string variable is used to instruct the computer to "PLAY" the contents of the string variable and then continue in the present string.
Dotted notes	The "Period" character "." is used following notes to instruct the computer to increase the length of the note by one half its normal value. If several "dots" are added to a note, each one will increase its note length by 1/2 its normal value.

Suffixes

There are four suffix characters that can be used to alter the values for Octave "O", Volume "V", Tempo "T" and Note length "L". The suffixes can be used to adjust the values of these commands without having to add numbers.

+ Adds one the the current value.
- Subtracts one from the current value.
> Multiplies the current value by two.
< Divides the current value by two.

For example: to increase the current tempo by one, you could use T+, or to decrease the volume for a few notes you could use V< to lower it by half and then use V> to restore it back to normal.

```
Example:  PLAY "T1; V5; P2; V10; A;P2; V20;A
          PLAY "XA$;XB$;XC$;XD$
```

The first example demonstrates a constant PLAY string that simply plays a note and increases the volume. The second example demonstrates the use of the Execute string command which is used to PLAY several pre-defined play strings. For more information and examples of using the PLAY statement, refer to the Extended Color Basic Manual chapter #9.

The Extended Hi-Resoulution Graphics statements in CBASIC-3 are almost identical to those in Enhanced Color Basic. Some brief descriptions of the statements are given to show differences and examples of their usage. For more information on these statements and graphics refer to the Extended Color Basic Manual.

In the description of the following High Resolution Graphics Statements the notations:

x specifies the X-coordinant (horizontal position) on the graphics display area and is a numeric constant or expressing from 0 to 639.

y specifies the Y-coordinant (vertical position) on the graphics display area and is a numeric constant or expressing from 0 to 191.

c specifies an available color code and is a numeric constant or expression from 0 to 15. This is optional in many statements; if omitted, the foreground color is used.

HMODE Statement *** Not available in Basic

Syntax: HMODE value

The HMODE statement sets the graphics resolution the same as the HSCREEN command except that it does not perform a clear screen when used. The HMODE value ranges from 0 to 4 with 4 being the highest resolution mode (640*192 2-color). For more information see the Extended Color Basic Manual.

```
Examples: HMODE 4
          HMODE 2
          HMODE P
```

The first example sets the graphics mode to 4 (640*192 2-colors). The second example selects mode 2 which is 320*192 in 16 colors. The third example simply sets the mode to whatever the value of the variable P is at the time it is executed.

HCOLOR Statement

Syntax: HCOLOR foreground, background

The HCOLOR statement allows you to change the graphics foreground and background colors (within the available choices). The "foreground" and "background" colors are numeric constants or variable expressions from 0 to 15, and represent the palette #.

```
Examples: HCOLOR 5,7
          HCOLOR 7,5
          HCOLOR FG,BR
```

The first two examples simply show constants being used for the foreground and background colors. The second example reverses the foreground and background colors from the 1st example. The third example shows variables being used for the color codes.

HSCREEN Statement

Syntax: HSCREEN value

The HSCREEN statement is used to select between the Text or Enhanced Hi-Res Graphics modes (320/640 modes). When this command is used it will automatically clear the Hi-Res screen. If you don't want the screen to clear use the HMODE command.

```
Examples: HSCREEN 1
          HSCREEN 4
          HSCREEN 0
```

HSET Statement

Syntax: HSET(x,y,c)

The HSET statement is used to set a single point on the graphics screen to a specified color. The x and y coordinants are used to specify exactly which position on the screen you want to set. The c argument is used to specify the palette color #.

```
Examples: HSET(0,0,8)
          HSET(128,96,8)
          HSET(X1,Y1,8)
```

The first example will set a dot in the top left corner of the screen. The second example will set a dot in the center of the screen. The third example demonstrates the use of variables for the x,y coordinants.

HRESET Statement

Syntax: HRESET(x,y)

The HRESET statement does the exact opposite of the HSET statement. It "resets" a dot in the screen to the background color. The x and y arguments are used to specify exactly which dot on the screen is to be reset. Notice that you don't have to specify the color with HRESET since the computer automatically uses the background color.

```
Examples: HRESET(128,96)
          HRESET(X1,Y1)
```

The first example will reset the dot at the center of the screen and the second example demonstrates the use of variables for the coordinants.

HPOINT Statement

Syntax: HPOINT(x,y)

The HPOINT statement is similar in form to the HRESET statement, but instead of reseting the specified dot on the screen, it tests the color of a specified graphics point. Your program may then use the information any way you choose. The HPOINT statement returns a value from 0 to 15 to represent the color Palette slot of the specified graphics point.

```
Examples: C=HPOINT(128,96)
          IF HPOINT(X1,Y1) = 8 THEN 500
```

The first example will get the value of the color from the point in the center of the screen and assign that value to the variable C. The second example demonstrates the use of HPOINT in an IF/THEN statement, that is testing to see if the point at location X1,Y1 is orange in color, if so it will transfer control to line 500.

HCLS Statement

Syntax: HCLS color

The HCLS Statement is used to clear the graphics screen to a specified color 0-15. If a color is not specified, the screen will be cleared to the background color. This serves the same function for Hi-Res graphics as CLS does for the text screen.

Examples: HCLS
HCLS 6

The first example would simply clear the screen to the background color. The second example would clear the screen with the color "cyan" (color code 6).

HLINE Statement

Syntax: HLINE(x1,y1)-(x2,y2),a,b

The HLINE statement is used to draw a line, box or rectangle on the graphics screen. The x1,y1 coordinants are used to specify the starting point on the screen and the x2,y2 coordinants are used to specify ending point for the line. The line is then drawn by the computer between these two points. The "a" argument is used to tell the computer whether to draw the line using the pre-specified foreground color (PSET), or to use the pre-specified background color (PRESET). The PRESET function may be compared to "erasing" rather than drawing on the screen, since the background color makes the line invisable.

The "b" argument is an option that allows you to draw a "Box" or rectangle without having to draw four separate lines. All you have to do is specify two of the opposing corners for the square in x1,y1 and x2,y2, and add ",B" to the statement. You also have the option to add an "F" to the optional argument ",B" to produce ",BF". This will let you "fill" the box with the foreground color to produce a solid box.

Examples: HLINE(0,0)-(255,191),PSET
HLINE(64,48)-(96,64),PSET,BF

The first example will draw a line from the top left corner of the screen to the bottom right corner. The second example will draw a rectangle 32 points across and 16 points down in the middle of the screen and fill the box with the foreground color.

HPAINT Statement

Syntax: HPAINT(x,y), color, border color

The HPAINT statement allows you to "paint" any shape with any available color. The x,y coordinants are used to specify where on the graphics screen the painting is to begin. The "color" parameter specifies the color code of the paint 0-15. The "border color" parameter tells the computer the color code of the border at which the painting is to stop. If the computer reaches a border other than the specified color, it will paint over that border.

Examples:
```
10 HMODE 3,1
20 HCLS
30 HSCREEN 1,1
40 HCIRCLE(128,96),90
50 HPAINT(128,96),8,8
60 GOTO 60
```

The sample program will draw a circle in the center of the screen and paint in orange.

HCIRCLE Statement

Syntax: HCIRCLE(x,y),r,[color],[hw ratio],[start],[end]

The HCIRCLE statement will allow you to create a full circle, a partial circle or an ellipse using a single Basic statement. The only arguments required to make a circle are the center point coordinants (x,y) and a radius "r", all other arguments are optional. The radius "r" specifies the circle's radius in units from 0 to 255, each unit of measurement is equal to one point on the screen. The optional "color" specifies an available color 0-15, default is the foreground color. The height/width ratio "hw" is optional, it specifies the ratio or the circle's "width" to it's "heigth", if not specified, a value of 256 is used (1:1). A value less than 256 results in a circle "wider" than it is high, a value over 256 results in a circle "Higher" than it is wide. The start & end options allow you to draw just part of a circle (an arc). To use this option, specify the point where the arc is to begin (0-255), insert a comma, and then the point where it is to end (0-255). The starting point (0) for any circle is equivalent to 3 o'clock on a clock, 64 would be 6 o'clock, 128 would be 9 o'clock and 192 would be 12 o'clock. To use the start and end options, you must specify the "hw" ratio, for a normal arc, use hw=256. For more information on the "HCIRCLE" statement refer to the Extended Color Basic Manual.

```
Examples:  HCIRCLE(128,92),95
           HCIRCLE(128,92),30,1,256,64,192
           HCIRCLE(X1,Y1),30,1,HW,ST,EN
```

The first example demonstrates a simple circle drawn at the center of the screen. The second example demonstrates the use of all options to draw a half circle from 6 o'clock (64) to 12 o'clock (192). The last example is similar except variables are used instead of constants.

Please note that the "hw", "start" and "end" arguments in CBASIC-3 differ from those in Color Basic since they are fractional numbers. If these items are specified as constants in the HCIRCLE statement the normal Color Basic decimal format will be accepted by the compiler. When variables are used the values assigned to the variables must conform to the specifications listed above for "hw", "start" and "end".

HPRINT Statement

```
Syntax:  HPRINT (x,y),String
```

The HPRINT command allows you to print a text message on the Hi-Res (320/640 by 192) screen. The x and y positions are a column (0-39 for 320 modes or 0-79 for 640 modes) and a line position (0 to 23). The string is any valid string literal or variable up to the remaining character positions on the display.

```
Example:  HPRINT (10,12),"HELLO"
          HPRINT (X,Y),A$
```

HDRAW Statement

Syntax: HDRAW string expression

The HDRAW statement is used to draw a line or series of lines, by specifying its direction, angle, and color. The string expression may be a "literal", string variable or expression used to contain the HDRAW statement commands. The HDRAW commands are as follows:

Motion cmds.	M = Move the draw position
	U = Up
	D = Down
	L = Left
	R = Right
	E = 45 degree angle
	F = 135 degree angle
	G = 225 degree angle
	H = 315 degree angle
	X = Execute a substring & return
Mode cmds.	C = Color
	A = Angle
	S = Scale
Option cmds.	N = No update of draw position
	B = Blank (no draw, just move)

The Motion commands tell the computer where to start drawing on the screen (Mx,y), which direction to draw in (U,D,L,R,E,F,G,H), and how many dots to draw (U25,D25,E30,.. etc.). The motion command Mx,y; positions the cursor to a specified x,y point on the screen, to avoid unwanted lines on the screen preface the M command with the letter B (BM 128,96). The M command can also specify a position "relative" to the current x,y position by preceeding each of the coordinants with a "+" or "-" sign (BM+15,-15).

The Mode command "Cx" allows you to specify a color code 0-15 to be used while drawing (C7).

The Mode command "Sx" allows you to "scale" a drawing up or down, where x is a number from 1 to 62 to indicate the scaling factor in 1/4 units. A scale of 4 = full scale 4/4, a scale of 1 = 1/4 scale, a scale of 8 = double scale 8/4 and so on up to 62. After an Sx command all motion commands will be scaled accordingly until the next Sx command.

The Mode command "Ax" allows you to specify the angle at which a line is to be drawn, 0 = 0 degrees, 1 = 90 degrees, 2 = 180 degrees, and 3 = 270 degrees. All lines drawn following an Ax command will be drawn relative to the angle displacement specified by Ax.

The option "B" blank, has already been mentioned in relation to the Move command. It can also be used to preceed any motion command to cause a blank line to be drawn. This only affects the line immediately following the "B" blank option.

The option "N" can be used to tell the computer "not" to update the x,y location after drawing a line, but to return to the current x,y location before doing the next command. This only affects the command immediately following it.

The last Motion command "X" allows you to execute another HDRAW string assigned to the string variable immediately following the command (XA$). The computer will execute this HDRAW string and then return to the next command following.

```
Examples:  HDRAW "BM128,96;E25;F25;G25;H25"
           HDRAW A$
```

The first example moves the draw position to the center of the screen 128,96 and draws a box. The second example shows the use of a string variable for the HDRAW string. For more information and examples of using the HDRAW statement refer to the Extended Color Basic Manual chapter 7.

HGET & HPUT Statements

```
Syntax:   HGET (x1,y1)-(x2,y2), buffer#
          HPUT (x1,y1)-(x2,y2), buffer#, action
```

The HGET and HPUT statements are used to "get" a rectangular area which contains a graphics display, store it in a buffer, then "put" the buffer back on the screen at a later time. The x1,y1 and x2,y2 coordinants are used to tell the computer where the upper left corner and lower right corner of the rectangular graphics area is located on the screen to HGET or HPUT. The "destination" for HGET is the name of a pre-defined HBUFF # used to store the graphics data.

The "options" for the HPUT statement determine how the data is to be written to the display. They consist of the following:

PSET — Set each point that is set in the source buffer.

PRESET — Reset each point that is set in the source buffer.

AND — Logically AND each point in the buffer with each corresponding point in the destination rectangle. If both points are set then the screen point will be set, otherwise reset.

OR — Logically OR each point in the buffer with each corresponding point in the destination rectangle. If either point is set then the screen point will be set.

NOT — Reverses the state of each point in the destination rectangle reguardless of the HPUT buffers contents.

Before using the HGET or HPUT statements, a buffer must be defined to store the graphics data. The size of the buffer is determined by the size of the display rectangle. It must be large enough to hold all the data, but not too large, or memory space will be wasted. Since CBASIC-3 uses the same buffer area as Enhanced Color basic the calculations are the same, see page 173-175 in your Co-Co 3 Extended Book for more information.

Examples:

```
10 HBUFF 1,43
20 HGET (10,0)-(20,10),1
30 HCLS
40 HPUT (100,100)-(110,110),1,PSET
50 GOTO 50
```

The sample program simply "gets" the 10 by 10 rectangle from the screen and stores it in buffer 1. It then clears the screen and "puts" the rectangle at a different location on the screen.

HBUFF Statement

Syntax: HBUFF buff#, size

The HBUFF command is used to reserve memory space to store a rectangle of graphics information for the HGET & HPUT Statements. The buff# is a number that labels the buffer for use with HGET or HPUT Statements and the size is the number of bytes to reserve. For more information on determining the size for the HBUFF command see the Extended Color basic book chapter 31, page 173.

Example: HBUFF 1,43

BORDER Statement *** Not available in Basic

Syntax: BORDER value

The BORDER command allows you to select a pallette number 0-15 for use as the BORDER color on the screen. You can change colors for the border at any time. Normally the Border color is selected the same as the background color when a HCLS command is executed. The BORDER command allows you to select a new color without having to clear the screen.

Example: BORDER 14

PALETTE Statement

Syntax: PALETTE reg#, color
PALETTE RGB
PALETTE CMP
CMP
RGB

The PALETTE command is used to select any of the available 64 colors for a specified palette register (0-15) number. If RGB or CMP follows the PALETTE command it will cause the default colors for an RGB monitor or Composite monitor to be used. The RGB or CMP commands can also be used by themselves to obtain the same results.

Example: PALETTE 3,44
PALETTE CMP
PALETTE N,C

INPUT Statement

Syntax:
```
INPUT var, var,..., var
INPUT "literal string"; var,..., var
INPUT #N, var, var,..., var
```

The INPUT statement causes code to be generated which displays a "?" prompt and space on the screen or RS-232 device. It then reads characters into the input buffer until 255 characters have been read or an ENTER or BREAK key depressed. A carriage return is output to the screen or RS-232 device when the last character is input. During the entry of data, each character input is echoed back to the screen or RS-232 device.

At run-time, entry of a shift/left arrow will delete the current buffer contents. A left arrow will backspace the cursor and erase the character.

If a "literal string" immediately follows the INPUT statement, that string of characters will be displayed on the screen or RS-232 device before the "?" prompt.

The variables specified may be numeric or string, subscripted or simple type. When the program is "looking for" a number from the current position in the input buffer, it will skip leading spaces, if any, and read a minus sign (if any), and up to five number characters. The numeric field is terminated by a space, comma, or end of line. If a non-digit character is read or any other illegal condition, a value of zero will be returned for the number. The symbols "$" and "&H" may also be used to input hexadecimal numbers directly.

If a string-type field is being processed, leading spaces will be skipped unless enclosed within quotes "" and data accepted, until the variable field is terminated by a comma, end of line, ending quote, or when the string variable is "full". If no characters are available, a null string will be returned.

The INPUT statement may also be followed immediately by the "#" pound symbol and a device number or numeric variable. When doing input from a device such as Tape or Disk, the file must have been previously "opened" by the OPEN statement, or an error will occur. This condition is detectable by the ON ERROR GOTO statement. For more information on device I/O see the section on TAPE & DISK I/O.

```
Examples:  INPUT A,B,AX$,RA$(N)
           INPUT #-1, A$,N,A(4,N)
           INPUT "Enter your name";NA$
           INPUT #-3,"Enter your name";NA$
           INPUT #N,A$,B$,C,D
           INPUT N
```

LINEINPUT Statement

```
Syntax:    LINEINPUT string variable
           LINEINPUT "literal string"; string variable
           LINEINPUT #N, string variable
```

The LINEINPUT statement is almost identical to the standard INPUT statement, except it assigns the entire contents of the input buffer to a string variable, including commas, spaces and quotes. Only one variable name may be listed since any following variables will be assigned a null string. When used for keyboard or RS-232 input, it will not display the "?" prompt.

As in the standard INPUT statement a "#" pound sign followed by a device number or numeric variable may be used immediately following the statement to direct input from tape, disk or the RS-232 port.

```
Examples:  LINEINPUT A$
           LINEINPUT "Enter your full name";NA$
           LINEINPUT #-1, AX$
           LINEINPUT #-3, "Enter your Name ";NA$
```

PRINT Statement

Syntax:
```
PRINT output spec[,;] ... output spec
PRINT @N, output spec[,;] ... output spec
PRINT #N, output spec[,;] ... output spec
```

The PRINT statement is used to output information to the screen, printer, RS-232 port, tape or disk. The output spec's are processed and the appropriate characters are put in the I/O buffer. The buffer is then output to the proper device.

The output spec's may consist of string or numeric expressions, or the output function TAB(expr) which inserts spaces in the buffer until the position "expr" is reached. Each item in the list is separated by a delimiter which is either a comma or semicolon. The buffer is divided into thirty-two 8 character zones, which are effectively tab stops every eighth position. If a comma is used as a delimiter, the next item will begin at the first position of the next zone. If a semicolon is used, NO spacing will occur. A semicolon at the end of a PRINT statement will inhibit the printing of a carriage/return at the end of the line. A PRINT statement without any output spec's will produce a carriage return only.

The PRINT statement can optionally be followed by the "#" pound sign and a number or numeric variable to direct output to a device other than the screen. If output is attempted to tape or disk, a file must have been previously "opened" for output or an error condition will occur. This can be detected at run-time by the ON ERROR GOTO statement. For more information refer to the section on TAPE & DISK I/O.

The PRINT statement can also be followed by the "@" symbol and a number or numeric variable to print at a specified location on the screen. If the standard screen is being used, the highest location available is 511 (32 by 16). If the HIRES option was used in the program, the highest location can vary from 671 to as high as 6119 depending on the selected line length.

Examples:
```
PRINT A,B,C

PRINT A$(N),A$(N+1)

PRINT #-2, A,A$,B,B$,NA$

PRINT #-3, "Hello ";NA$

PRINT @12,"Hello ";NA$

PRINT #N, A$;TAB(N+M);BA$
```

INKEY Statements

```
Syntax:   INKEY <numeric var.>
          INKEY$ <string var.>
```

CBASIC-3 allows the use of an INKEY type function to return a numeric value or a string value. This can be very helpful in evaluating data returned to represent a key pressed, since it is normally converted from a string to a numeric value before processing. It also requires much less code to evaluate a numeric value in an IF/THEN statement, than to evaluate a string argument. It also generates less code and executes faster than doing an INKEY$ function.

```
Example:  100 A=INKEY
          200 IF INKEY=13 THEN 500 ELSE 200:'WAIT FOR ENTER
```

RS-232 PORT Device #-3 support for:

```
INKEY #-3                    INKEY$ #-3
INPUT #-3                    LINEINPUT #-3
PRINT #-3
```

CBASIC-3 supports the RS-232 port on the back of the CoCo for input and output, using standard basic commands and functions. All commands work the same as they do normally except for the INKEY and INKEY$ functions. When an INKEY type function is executed on the RS-232 port (device #-3), it will scan the port for input for approximately 2 seconds waiting for a character. If no character is received within that time limit a 0 or null string value is returned. If received data is available it will return the data as soon as a full character is received.

```
Example:  INPUT #-3,variable list
          PRINT #-3,variable list
          A=INKEY #-3
```

Printer & RS-232 Baud rate Statements

Syntax: BRATE = baud rate
PRATE = baud rate

These commands are used to set a desired Baud rate for the Printer (PRATE) or the RS-232 port (BRATE) from within a compiled CBASIC-3 program. The value must be a rate between 110 and 9600, variables or numeric expressions are not allowed. Valid baud rates are: 110, 300, 600, 1200, 2400, 4800 and 9600. A baud rate of 110 is not valid for use on the RS-232 port. If you are going to run the computer at double speed select a rate that is half the desired baud rate, ie. to select 4800 baud use BRATE=2400.

Example: PRATE=9600
BRATE=1200

Position @ Statement

Syntax: POS@

The POS@ function has been added to allow access to the current print @ position on the screen. It will return the current print @ position for either the standard 16*32 screen or the HIRES screen if the HIRES option was used. This can be handy when you wish to display a message on a different part of the screen than the current cursor location such as a status update. Then return to the original cursor position for input or another display.

Example:

```
100 A=POS@
110 PRINT @0,"TIME IS RUNNING OUT";
120 PRINT @A,"";
```

PUTCHAR Statement *** Not available in Basic

Syntax: PUTCHAR device#, #variable or value

The PUTCHAR command allows you to send or write a single byte value of information to a specified device. You can use any numeric expression, variable or number for the value to be output. The data will be output as a single byte to the device which means that only the least significant byte of a variable or expression will be used (0-255) with the most significant byte discarded. This can be very useful for doing screen dumps or outputing binary data to a device or file. This command will allow you to overcome CBASIC's limitation of not being able to send a null (00) character out as part of a string or string variable (00 is used as an end of string marker). Any legal device number can be used (-3 thru 9) to select where the output data will sent. If a device number is not specified the Screen will be used (device 0).

```
Example:   PUTCHAR #-2,A
           PUTCHAR #-2,PEEK(A+B)
           PUTCHAR DV,223
```

GETCHAR Statement *** Not available in Basic

Syntax: GETCHAR device#, numeric variable

The GETCHAR command allows you to get or input a single byte of information from a specified device. In some ways it is similar to the INKEY statement except that when used to get a byte from the Keyboard or Serial port it will wait until a byte is received or key pressed before continuing on to the next statement. The byte returned from the command is always stored in a numeric variable as a value between 0 and 255. For example if a GETCHAR command was used to input a value from the keyboard and the "A" key was pressed a value of 65 or $41 would be stored in the variable specified. It can also be useful to read binary information from a disk file or binary serial data from the RS-232 port such as in an XMODEM file transfer etc. If a device number is not specified it will default to the keyboard (device 0).

```
Example:   GETCHAR #-3,A
           GETCHAR B
```

Disk and Tape I/O in CBASIC-3 is channel oriented meaning a file to be used for input or output must be "opened" and assigned a channel number by which all further operations on that file are performed. CBASIC-3 supports up to 9 Disk channels (1-9) and 1 tape channel (-1), which are maximum number of files that may be open at any time.

All disk and tape file names are defined the same as the normal Basic operating system's. All files used by CBASIC-3 are standard ASCII formatted data files.

Many of the CBASIC-3 disk and tape operations are the same as the normal Basic, so information as to disk and tape operations in the Basic Reference manuals will generally apply.

NOTE: In the descriptions of disk and tape statements that follow the term "file-id" refers to an 8 character file name. For disk files the 3 character extension and drive number may also be included. If a file extension is not included, a ".DAT" extension will automatically be assumed. If a drive number is not specified, it will default to drive #0.

The term "#F" refers to a channel number which may be a numeric constant or variable for reference to a specified disk channel, or tape. It is up to the programmer to make sure that any variable used for a channel number is within the correct range, and that the device has been previously "opened" by the OPEN statement. All errors are detectable at run-time by the ON ERROR GOTO statement. If an error should occur during disk or tape I/O and ON ERROR trapping is disabled, unpredictable results can occur.

Remember that Disk file buffers can not reside in the upper 32K of memory space. You can determine this from the Variable table listed at the end of the program when compiled. Disk file buffers are shown as "#n" where n is the file number as used in the program. Fielded record buffers are shown as "*n" where n is the file number it is associated with in the program. If the address of the next variable is greater than 8000, it means that the associated disk buffer or record is in the upper 32K of memory space. If this condition exists you can use the BASE statement in the beginning of the program to assign variables in the upper 32K of memory and when you get to the first statement that references a disk file buffer, change the base back to zero. This will usually do the trick.

FILES command

The FILES command is recognized by CBASIC-3 to avoid syntax errors and confusion when converting Color Basic programs to CBASIC. Since CBASIC-3 dynamically allocates file buffers as files are created, the FILES command has no function and does not generate any program code. When encountered in a CBASIC-3 program, the FILES statement is treated the same as a REM statement.

AUDIO ON/OFF

The AUDIO ON/OFF command is used to either connects or disconnects the sound coming from the cassette tape recorder to the T.V. speaker. The AUDIO command must be followed by either the word "ON" to enable sound to the speaker, or "OFF" to disconnect the cassette sound from the speaker.

```
Example:    AUDIO ON
            AUDIO OFF
```

MOTOR ON/OFF

This command allows the user to manually turn the cassette recorder motor either on or off under program control. Normally the cassette tape recorder is controled automatically when reading or writing tape files.

```
Example:    MOTOR ON
            MOTOR OFF
```

OPEN Statement

Syntax: OPEN "I/O/R/D",#(1-9),"file-id",(record length)
OPEN "I/O",#-1,"file-id"

The "OPEN" statement for Disk & Tape files is almost identical to the standard Basic Open command. Basic normally allows file numbers -2 and 0 to be used in association with Printer and Screen or Keyboard Input and Output. On a Disk System, the system allows file numbers between one and nine (1-9) to be used in association with disk files. These files can be opened for Input, Output, Direct, or Random access. Tape files (-1) can only be opened for Sequential access, Input or Output only. If a disk file is opened for Input or Output, it can only be accessed in a sequential manner, examples of these would be text, program or cassette tape files. They must be either read or written to in sequence, and cannot be accessed in any other manner. Random or Direct access files can be read or written to in any portion of the file, and will be discussed in more detail further on in this manual.

The file id can be a string or string variable. For disk files, it must include a drive number if other than drive 0. The disk file type will default to a data file ".DAT". On disk files, a record length may be specified following the file-id for Random or Direct access files, if not a default record length of 256 bytes is used.

NOTE: In the OPEN statement the channel number "MUST" be a constant number in the range of 1-9 for disk, or -1 for tape.

Examples:

```
10 OPEN "I",#2,"LABLES.TXT:1"
210 OPEN"R",#1,"DATABS:2",128
610 OPEN"I",#-1,"DATABASE"
```

The first example shows that a sequential input file (file #2) is to be opened on drive #1, and the file will be called "LABEL.TXT". The second example shows a Random access file (#1) will be opened on drive #2, and the file name is "DATABS.DAT" (it's record length is 128 bytes). The third example will open the Tape file "DATABASE" for input.

When a Random access file is opened and the file is not on the specified disk drive, or does not exist, a file will be created with no data in it. If a file is to be opened for Input and does not exist, an error will be reported. If a file is to be opened for Output and already exists on the disk, it will automatically be "KILLED" or Scratched and no warning or message will be displayed.

PRINT Statement

Syntax: PRINT #F,(VARIABLE LIST)

The PRINT statement is used to output data sequentially to a disk or tape file buffer. It can be used for sequential or random access disk files (you may use a comma or semi-colon to format or separate each item). Normally when using the PRINT statement the data is output to the file in the same exact format as it would be output to the Screen or Printer. This includes spaces output by TAB functions or by commas, etc. This may not be exactly what you want if you plan to read the data back out of the file with an INPUT statement. Normally you would use the "WRITE" statement if you want data to be in this format. You can also make the PRINT" statement work the same by using a "#number" instead of a #variable for the device number.

Example:

```
10 OPEN"O",#1,"NUMBER"
20 FOR I=1TO 100
30 PRINT #1,I
40 NEXT I
50 CLOSE
```

This example would write the numbers from 1 thur 100 to a disk file on drive 0 with the name "NUMBER.DAT". Each number would be separated by an "enter" character in the file. If a semi-colon were used following the "I" such as "PRINT #1,I;", the numbers would be written with only a single space between each one. Also if a comma were used to separate two items such as "PRINT #1,I,I+1", then there would be several spaces between "I" and "I+" followed by an "enter" character.

WRITE Statement

Syntax: WRITE #F,(VARIABLE LIST)

The WRITE statement is used to output data sequentially to a disk file buffer. It can be used for sequential or random access files (you may use a comma "ONLY" to separate each item). When using the WRITE statement the data is output to the file with delimiters between each item as it is written to the file. This is exactly what you want if you plan to read the data back out of the file with an INPUT statement. Normally you would use the "WRITE" statement if you want data to be in this format. The WRITE statement can be used with random access files by following each WRITE statement with a "PUT" statement (see GET & PUT for the format). If an attempt to write more data than the record buffer can hold is made an error will be reported. Note, the record buffer does "NOT" have to be Fielded when I/O is performed using WRITE, PRINT and INPUT in this format.

Example:

```
 10 OPEN"O",#1,"DATA"
 20 A$="JOHN SMITH"
 30 B$="TEST DATA"
 40 C=9875432
 50 WRITE #1,A$,B$,C
*55 PUT #1,1
 60 CLOSE
```

This example would write the data "JOHN SMITH", "TEST DATA" and the number "9875432" to the disk file on drive #0 called "DATA.DAT". Each item would be separated by a delimiter character in the file. Line 55 is there to show how WRITE would be used to store data in a random access file.*

INPUT Statement

Syntax: INPUT #F,(VARIABLE LIST)
LINE INPUT #F,(VARIABLE LIST)

The INPUT statement like the Write and Print statements can be used to communicate with a sequential or random access disk file. The variable list is the same as the normal INPUT statement for Tape or Keyboard I/O. Each item in the variable list is separated by a comma, and can be mixed string and numeric variables as long as the input data from the disk file is the same type. Numeric data can be read into a string variable as long as it was not created using the MKN$ function. The LINE INPUT command functions identically, except it will ingore delimiters such as commas, quotation marks and colons, Everything is accepted. (See the Extended Basic manual for further details).

Example:

```
5  DIM A(100)
10 OPEN "O",#1,"NUMBER"
20 FOR I=1 TO 100
30 WRITE #1,I
40 NEXT I
50 CLOSE #1
60 OPEN "I",#1,"NUMBER"
70 FOR I=100 TO 1 STEP -1
80 INPUT #1,A(I)
90 NEXT
100 CLOSE
```

The example shows a file being written with the numbers from 1 to 100, and then being Rewound (CLOSED & RE-OPENED for INPUT). The file is then read storing the data in the array "A(100)" in reverse order.

EOF Function

Syntax: EOF(#F)

The EOF function is used to determine whether the file number (#F) specified is at the End Of File during a read. It will return a value of 0 if there is more data to be read in the file, and a -1 if there is "no" more data. The EOF function should be used prior to every INPUT performed on a file, or an "IE" (Input Past End Of File) error will be reported when the end of file is reached. (See Color Basic manual for more information)

Example:

```
10 OPEN "I",#1,"LABLES"
20 IF EOF(1) = -1 THEN 50
30 INPUT #1,A$,B,C$
40 PRINT A$,B,C$
45 GOTO 20
50 CLOSE #1
```

The Example shows how the EOF function is used in a program to test for an end of file condition prior to each INPUT command. If line 20 was not in the program an "IE" error would be reported when the end of file is reached.

CLOSE Statement

Syntax: CLOSE #F
CLOSE

The CLOSE statement is used to terminate I/O between a Basic program and a disk, or tape file, whether for Input, Output or Random/Direct access. Closing a file will release the memory space used for the disk file sector buffer (FIB), and random access sector buffer if used. This statement can have two forms; one of which specifies a file number previously used in an OPEN statement to be closed and the second form is used without any file number and specifies that all open files are to be closed. It is very important to CLOSE files when communications is finished so that all information is written to the correct disk or tape file, and on the correct disk before it is removed from the drive (SEE Color Basic manual for further details).

Examples:

```
150 CLOSE #1
240 CLOSE
```

ERR & ERNO Function

Syntax: ERR
ERNO

The ERR & ERNO functions works in conjunction with the ON ERROR GOTO statement. The ERR function allows access to the last error reported in general or on any active file number. See the example following the ON ERROR GOTO statement.

ERL & ERLIN Function

Syntax: X=ERL
X=ERLIN

The ERL & ERLIN functions also works in conjunction with the ON ERROR GOTO statement. These functions allows access to the number of the line in which the last error occured. For ERL & ERLIN to function properly the TRACE ON function must be enabled. Otherwise these functions will return a value of zero.

ON ERROR & ON ERR GOTO Statement

Syntax: ON ERROR GOTO line#
ON ERR GOTO line#

The ON ERROR & ON ERR statements allows the user to handle system errors without halting the current program execution, by passing control to a specified line number in the program to process the error. These functions can be changed at any time in the program to allow for a general error handling routine, or may be changed for a specific disk error handling not covered by the general error handler. If no error handling is specified, a normal basic error display & halt will occur. Error handling for any active file may be changed at any time or disabled by specifying a "GOTO" line number of "0". This can be used for disabling the general "ON ERROR" handling as well.

Example:

```
10 ON ERROR GOTO 300:TRACE ON
20 OPEN "I",#1,"NAMES1"
.
.
300 PRINT "ERROR #":ERR;" IN LINE NUMBER ;ERL
```

FIELD Statement

Syntax: FIELD #F, length AS var,...., etc.

The FIELD command is used in conjunction with Random or Direct access files to format a disk record into specific fielded variables. By fielding a file record (buffer), the system associates specified areas in the disk file record to variables. When fielding a record, the total length of the the fielded areas may not exceed the length of a single record as defined by the OPEN statement. Each time a FIELD statement is executed, the record is fielded starting at the first position of the record, therefore many different variables may be associated with the same area or overlapping areas in a record. The record may be fielded at any time during program execution, provided the associated file number is open for random or direct ("R" or "D") access. Once a variable has been fielded, its value will be whatever data is currently in the associated record buffer. When the data is changed by use of the "GET" statement, the variable data will also change according to the file contents.

When the FIELD statement associates a string variable with an record buffer in Color Basic, it can only be assigned data via the RSET or LSET statement. This is not the case in CBASIC. Once a variable is assigned to a fielded record buffer it cannot be reassigned. Therefore in CBASIC-3 the "LET" or implied "LET" statement can be used on fielded variables. Since variables are assigned in fixed locations, you cannot use a previously fielded variable in another FIELD statement. You can however field a file record more than once, as long as successive FIELD statements use different variable names. Data can also be moved to a fielded variable by the use of the LSET or RSET statements. The following examples will show some methods for fielding a record.

Example #1:

```
10 OPEN "R",#1,":1 TESTER.TXT",64
20 FIELD #1,32 AS A$,32 AS B$
30 FIELD #1,10 AS IT$,12 AS VN$,20 AS DI$,22 AS CO$
```

EXAMPLE #2:

```
10 OPEN #1,"R","TESTER.DAT",256
20 FIELD #1,20 AS FIRST$,20 AS LASTN$,40 AS ADDRES$,
15 AS CITY$,2 AS STATE$,5 AS ZIP$,40 AS COMPANY$
```

The first example shows how a file record can be fielded more than once with different variables. The second example shows a mailing record be defined and that the entire record length does not have to be fielded.

RSET & LSET Statements

Syntax: RSET var = expression
LSET var = expression

The FIELD statement has previously been used to assign a string variable name to a portion of the disk file record buffer. These and other string variables can receive their data via the RSET and LSET commands so that the unused string storage will be filled with spaces. These commmands store the result of a string expression into the variable space either right justifed (RSET) or left justified (LSET). If the transfered data lenght is less than the length of the variable space allocated, the unused spaces will be filled with "space" characters. If the transfered data is larger than the fielded variable it will be truncated or lost.

Example:

```
10 FIELD #1,10 AS A$,10 AS B$
20 LSET A$="TESTING"
30 RSET B$="FIELDSET"
40 LSET B$=STRING$(10,32)
```

```
Location 1 2 3 4 5 6 7 8 9 10   "-" = SPACE

A$ field T E S T I N G - - -     LSET (LEFT JUSTIFIED)

B$ field - - F I E L D S E T     RSET (RIGHT JUSTIFIED)

B$ field - - - - - - - - - -     FILLED WITH SPACES
```

GET & PUT Statements

Syntax: GET #F,(RECORD #)
PUT #F,(RECORD #)

The GET & PUT statements are used in conjunction with Random or Direct access files only. They tell the system to read or write the "next" or specified record# of the designated file. The correct disk sector is computed and read into the sector buffer if necessary. The correct portion of that sector information is then transferred to or from the file record buffer. Once there by the use of a GET statement, the data can be manipulated or transferred from a fielded variable or assigned to a different variable by the use of an INPUT or LET statements. If a PUT statement is executed, the record buffer is transferred to the disk file sector buffer and written to the disk when necessary. A variable can be used to specify the file and/or record number to PUT or GET. When no record number is specified, the next record number in sequence will be used. If an attempt is made to GET a record that is past the end of the file, an error will be reported. If an attempt is made to PUT a record past the end of file, the file will automatically be expanded to store the record with extra space allocated for future expansion. If no disk space can be obtained for file expansion, an error will be reported.

Example:

```
10 OPEN "R",#1,"DATA",128
20 INPUT"HOW MANY RECORDS TO INITALIZE";RECORD
30 REM INITALIZE SPECIFIED NUMBER OF RECORDS
40 FIELD #1,INITALIZE$ AS 128
50 LSET INITALIZE$="EMPTY RECORD"
60 PUT #1,RECORD:'REM ALLOCATE RECORD SPACE
70 FOR I=1 TO RECORD
80 PUT #1,I
90 NEXT I
100 CLOSE #1
```

This example shows a random file being opened and the operator being prompted for the number of records that the file is to be initialized for, when input the highest record # is written first to expand the file. The rest of the records are then initialized in sequence via the for next loop until the highest record is re-written & the file closed.

CHAIN Statement

Syntax: CHAIN "file id.ext:drive",offset

The CHAIN statement allows Machine Language Disk programs to be loaded and automatically executed. It is identical to the format of the Basic LOADM command. CBASIC-3 will allow any machine language program to be loaded and executed in this manner, even if it loads right over the currently executing program in memory. However, if a program has an I/O error after it has been partly loaded into memory, unpredictable results may occur. The file name can be any valid string expression and the offset can be either a number or numeric variable.

```
Example:   10 CHAIN "BIOIA"
           349 CHAIN A$
```

The first example shows the command being used with a literal string to load and begin execution of the machine language program 'BIOIA.;" from drive 0 (default). The second example shows it being used in a program statement line where the variable 'A$' is being used to pass the drive and file id paramaters to the disk operating system.

KILL Statement

Sytax: KILL "file-id.ext:drive"

The KILL statement is the same as the Basic Kill command only it must be used in a basic program. The KILL command can specify only a single file on a specified disk.

```
Example:   10 KILL "TESTER.DAT:2"
           50 KILL DF$
```

The first example shows that the individual file called "TESTER.DAT" will be removed from the disk on drive #2. The second example shows the use of a string variable to specify the file to be removed.

RENAME Statement

Syntax: RENAME <old file-id> TO <new file-id>

The RENAME command is used to change the name of a specified file to a new name. If the Old file name does not exist or the New file name is already being used an error will be returned. Both the Old and New file specifiers can be either string variables or literals.

```
Example:  RENAME"TEST.BAS" TO "TESTER.BAS"
          RENAME A$ to B$
          RENAME A$ to "OLDFILE"
```

DSEARCH Statement

Syntax: DSEARCH(file.ext:drive)

*** Not available in Basic

DSEARCH is a numeric function that is used to determine if a specified file exists on the specified or default drive. The file id may be any valid String variable or literal. If the file does not exist a value of zero is returned, if the file does exist a value of -1 is returned.

```
Example:  IF DSEARCH(A$) THEN KILL A$
          A=DSEARCH(DATAFILE.DAT:2)
```

DRIVE Statement

Syntax: DRIVE <value>

The DRIVE command is used to specify a default Disk Drive for Disk I/O commands and functions. The Value can be either a number or numeric expression. There is no run time error checking for the Drive command, so a value greater than 3 is allowed (useful for 5 Meg. Hard Disk users). The default drive number is used when ever a Disk I/O command does not specify a drive number.

```
Example:  DRIVE 3
          DRIVE A
```

The first example would set the default drive to #3. The second examplw would set the default drive to the number specified by the variable A.

VERIFY Statement

Syntax: VERIFY <ON/OFF>

The VERIFY command is used to tell the system whether or not to verify (Read after Write) all write operations performed on the Disk System. The System normally leaves VERIFY OFF by default. IF VERIFY is enabled by the VERIFY ON command, all disk writes will take two disk revolutions to complete. One to write the sector and the next to read back the information written to verify that it was written correctly. It is a good practice to keep the verify option enabled to insure disk data integrity. However, it does take twice as long to write the same information on disk with VERIFY ON as it does to write it with VERIFY OFF.

Example:
```
VERIFY ON
VERIFY OFF
```

The first example would turn disk verification on (enabled) and the second example would be used to turn disk verification off (disabled).

DSKI$ & DSKO$ Statements

Syntax: DSKI$ drive, track, sector, A$, B$
DSKI$ drive, track, sector, BUF$

The DSKI$ and DSKO$ commands are used to perform Disk Input (DSKI$) and Output (DSKO$) without the use of the Disk Operating System. These commands input and output directly to a sector (256 bytes) on a specified disk. The drive, track and sector values can be any numbers or variables, and specify where the disk I/O is to be performed. CBASIC-3 has two options for using these commands. Since a sector is 256 bytes and string variables can only be a maximum of 255 bytes in length, two string variables are required to hold the contents of a single sector. Each of these variables is to be 128 bytes each. If the variable names specified in the command are not previously used in the program, CBASIC-3 will automatically create two consecutive variables of the required length. If the variable names were previously used in the program, they must be a minimum of 128 bytes each or an error will be declared. The second option is to use the CBASIC-3 variable BUF$ which is the 256 byte run-time I/O buffer. If BUF$ is used, it is the only variable that is to be specified.

Since direct disk I/O can easily destroy a disk file or the disk directory, you should be very careful when using these statements. Only an experienced programmer who has a good working knowledge of the disk system should even attempt to use these commands.

Example: DSKI$ 0,17,3,A$,B$
DSKO$ 0,TK,SC,BUF$

These commands may also use subscripted variables for the sector data storage, however, the data is stored in 256 consecutive bytes starting at the first variable specified. When using a subscripted variable, only the first variable need be specified and must be dimmed for a length of 128 bytes or incorrect results will occur. For example if the array A$ were to be used it would be dimmed something like DIM A$(35,128). This would be sufficient space for 36 blocks of 128 bytes each (18 sectors or 1 track). The program to read a full track into the array would be something like the following:

```
10 DIM A$(35,128)
20 FOR S=0 TO 17
30 DSKI$ 0,17,S+1,A$(S*2)
40 NEXT S
```

CLOADM & LOADM

Syntax: CLOADM "file name",offset
LOADM "file name",offset

The CLOADM & LOADM commands are identical to the Color Basic CLOADM & LOADM commands. They allows you to load Machine Language programs from cassette tape (CLOADM) or DISK (LOADM) into memory. The "file name" can be any valid string expression and the offset value is optional. If used, the offset value may be any valid numeric expression. The offset value is added to the load address of the program, that address is then used for the location in memory where the program will be stored.

Example: CLOADM "TEST",$1000
LOADM NA$,OF

CSAVEM & SAVEM

Syntax: CSAVEM "file name",begin,end,exec.
SAVEM "file name",begin,end,exec.

The CSAVEM & SAVEM commands are used to save a machine language program or file in memory to either tape (CSAVEM) or Disk (SAVEM). The "file name" can be any valid string expression or string variable. The begin, end and execution addresses of the file are required and may not be omitted. They can be any valid numeric expression or variable. CBASIC-3 does not check the validity of the addresses at run-time, it is up to the programmer to check for address validity (begin not greater than end).

Example: CSAVEM"TEST",$2000,$3000,$2000
SAVEM NA$,BEGIN,END,BEGIN+12

The following section will discuss the functions available for use with disk related I/O. All the functions listed will return a numeric value related to a particular disk file or drive. These functions can be used wherever a number or value is used in an expression.

FREE Function

Syntax: FREE <drive #>

The FREE function returns the number of available or free granuls on a specified disk drive. If no drive is specified, a default drive of 0 is used.

Example:

```
5 PRINT FREE(1)
10 IF FREE(2) > 10 THEN 100 ELSE 200
100 OPEN "O",#1,":2 DATA"
.
200 PRINT"LOW DISK SPACE ON DISK DRIVE #2"
```

LOC Function

Syntax: LOC(#F)

The LOCation function returns the current number stored in a Random Access File buffer for a specified file number. If used on a sequential access file, it will always return a value of 0.

Example:

```
100 PRINT @18,"RECORD #";LOC(1);" BEING PROCESSED"
```

LOF Function

Syntax: LOF(#F)

The LOF function returns the highest record number of the specified random access file. If used on a sequential access file, unspecified results will occur. This function can be useful to avoid accessing past the end of file which will cause an error. This value is also used for various types of sort functions and hashing access techniques.

Example:

```
100 FOR I=1 TO LOF(1)
110 GET #1,I:REM READ RECORD OF FILE
120 NEXT I
130 REM NOW POSTITIONED AT END OF FILE
```

MKN$ Function

Syntax: MKN$(number/variable)

This function will convert a numeric variable or number into a 2 byte coded string for storage in a formatted or fielded disk file buffer. It is normally used in conjunction with a fielded variable so that numbers can be stored in a disk file, using a field length of 2 bytes to store any number up to 5 digits in length.

Example:

```
5 A = 23456
10 LSET B$=MKN$(A)
```

CVN Function

Syntax: CVN(string variable)

This function will convert a 2 byte coded string previously created by the MKN$ function back to a numerical representation. It can be displayed directly or assigned to a numeric variable.

Example:

```
5 PRINT CVN(B$)
10 A = CVN(B$)
```

DIFFERENCES BETWEEN CBASIC-3 AND COLOR BASIC PROGRAMS
&
OTHER HELPFUL HINTS

Even though a CBASIC-3 program may be designed to perform the same function as a typical Color Basic program, it may appear to execute differently. For example:

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT
```

If the above program is run under Color Basic you would simply see it display the number from 1 thru 10 on the screen. It would then display an "OK" message and stop. The same program compiled under CBASIC-3 may appear to execute differently, even though it doesn't. If you watch the screen very carefully, and don't blink your eyes when executing the compiled version of the same version program. You will see that it will also display the numbers from 1 thru 10 on the screen, however, it disappears almost immediately after displaying the numbers. The next thing you see is the Color Basic message, just like you do when you first power on the computer.

One of the reasons for this is, CBASIC-3 produces pure Machine Language programs, and under some circumstances it modifies the configuration of the normal Color Basic operating system. If the program were simply allowed to return control back to the Color Basic Operating System, it may "appear" to work ok. But, if you started to write a program or perform some disk operation, the system might crash, destroy a disk, or some other unpredictable results might occur. In order to avoid this situation the CBASIC-3 program forces the computer to do a "Cold Start" when it is finished. This may not be necessary in all cases, but, it is the safest and most reliable way to return control back to Color Basic. Almost all good Machine Language programs, which modify the operation of the computer in any way will perform this type of "Cold Start" when the program is finished.

From looking at the previous example, it may appear the this is an un-necessary precaution. But, on the other hand, most programs will not be this simple, if they were, there really would be no reason to compile them into Machine Language programs. CBASIC-3 has a large variety of commands and functions that allow it to do many things besides straight forward Basic programs. It has the capacity to provide a complete Operating System environment for programs, not available in Color Basic. Normally these functions could only be performed by an experienced Machine Language Programmer. Things like Interrupt Handling and using the upper 32K of a 64K machine. You can easily and quickly control or manipulate Hardware Devices such as the X-PAD, DELUXE RS-232 Program PAK and various other devices available from third party vendors. In normal Color Basic operating these types of devices is slow and cumbersome. CBASIC-3 also provides you with an

optional Hi-Resolution Text Display Package that is directly attached to your compiled program. With these types of advanced operations, you can not expect to be able to keep the Color Basic Operating System completely functional. Therefore it must "Cold Start" the system to insure that Color Basic is completely operational when the CBASIC-3 program is finished.

If you need to see the results of a programs display on the screen before the program exits back to Color Basic, use an INPUT statement just before the STOP or END Statement. This will allow you to see the display and simply hit the "enter" key when you are finished.

REMARK STATEMENTS

With CBASIC-3 programs, the REM or ' statements do not affect the compiled program size or execution speed in any way. They do not produce any code within the compiled program. By using REMark statements generously, it will enable you to improve the internal documentation and readability of your program, without affecting it's performance. It pays to write well documented programs that can be understood and modified easily, either by yourself or others.

GRAPHICS STATEMENTS

CBASIC-3 has the same Graphics Statements that are available in Extended Color Basic, and consequently CBASIC-3 uses many of the graphics subroutines available within the Extended Basic ROM. Since CBASIC-3 uses the same machine language code to generate the same functions as Color Basic the actual time to draw or display the graphics is the same. However in a CBASIC-3 program, the same Graphics statement will execute about 4 times faster than Extended Color Basic. We probably could have made it much faster by rewriting the run-time graphics package but the cost in memory would be tremendous, and the Graphics syntax would not be compatible or as extensive as Extended Color Basic's. The reason that CBASIC-3 executes graphics faster than Color Basic is that the compiled program does not have to lookup the command and variable locations each time a graphics statment is executed, this is where the real speed increase comes from in CBASIC-3.

When using Graphics statements in CBASIC-3, if you use numeric constants for the x,y coordinates or parameters, the generated code will be shorter and execute slightly faster than using variables. This applies to the statements: CIRCLE, LINE, PSET, PRESET, SCREEN, PMODE, COLOR, PAINT, PUT and GET. This format will save from 8 to 20 bytes of code in the compiled program for each statement using this method. By making use of subroutines wherever possible for duplicated statements, you will also reduce a programs size significantly.

DIFFERENCES BETWEEN CBASIC-3 AND COLOR BASIC PROGRAMS
&
OTHER HELPFUL HINTS

USING SUBROUTINES

Since CBASIC-3 is a native compiler (generates actual machine code), each statement compiled will generate the equivalent machine code to perform that function. In many programs the same statement may be executed several times, especially in graphics programs. Each statement compiled will produce roughly the same amount of code, so even a single line which is used repeatedly in a program will produce a significantly larger program. If statements which are used repeatedly are made into subroutines and called with a GOSUB statement, the program size will be reduced significantly. A GOSUB statement only generates 3 bytes of code to call a subroutine, and to make a single statement or group of statements into a subroutine only requires that it be ended with a RETURN statement (1 byte of code). If this method is used to replace a single complex IF and/or THEN ... ELSE ... statement you could easily save up to 200-300 bytes of code for each occurence. A typical graphics or string statement using variables can use anywhere from 20 to 50 bytes for each occurance. So you can easily see how much memory space can be saved with little or no effect on program execution speed.

DATA & GENERATE STATEMENTS

Many Color Basic programs will use DATA statements to hold a machine language program or subroutine and then read the data and poke it into memory somewhere. It then calls the program or subroutine using the DEFUSR and USR statements or the EXEC statement. While this type of format can be used in CBASIC-3 it will waste a tremendous amount of program space since the DATA uses almost twice the amount of memory required, and it will still occupy space in the program after the proram or subroutine is poked into memory. CBASIC-3 has a statement called GENerate which allows machine language programs to be imbedded directly within the compiled program. This also allows these routines to be called from within the program by simply using a GOSUB or GOTO statement. The subroutine can return control back to the program by simply ending it with a RTS ($39) op code. This also eliminates the problem of placing the program in a part of memory where it will not be disturbed as well as using the READ and POKE statements to get it into memory.

DIFFERENCES BETWEEN CBASIC-3 AND COLOR BASIC PROGRAMS
&
OTHER HELPFUL HINTS

FOR/NEXT loops & TIMING

Many Color Basic prorams use FOR/NEXT loops for delays and timing. Since CBASIC-3 will execute a straight FOR/NEXT loop almost 1000 times faster it is not practical to use it for delays and timing. For example: FOR X=1 TO 1000:NEXT this statement in Color Basic will take almost 2 seconds to complete, but in CBASIC-3 is will be less than the blink of an eye. To generate accurate or consistent time delays in CBASIC-3 it is suggested you use the TIMER function, which will count up in 1/60 of a second intervals. Ex:

```
10 TIMER=0
20 IF TIMER <120 THEN 20
```

This will produce a delay of 2 seconds, if you know how many seconds you want to delay just multiply it by 60 and use that number in the IF TIMER statement. This format will also produce less code than the equivalent FOR/NEXT loop and will be much more accurate for timing and delays.

Get to know your Color Computer

If you have not had experience with the 6809's machine language, take the time to aquire some understanding of it. There are many good books and reference manuals available from Radio Shack. It is not absolutely necessary to have an understanding of machine language to use the CBASIC-3 compiler, in fact it was designed to be as compatible with the Color Basic interpreter as possible. This enables you to write and debug most programs using Color Basic, which is much easier than trying to debug machine language programs. However, many of the advanced features of CBASIC-3 can not be used in Color Basic. If you have a good understanding of how the machine works and operates, you will have much less difficulty using CBASIC-3 and its advanced features.

DIFFERENCES BETWEEN CBASIC-3 AND COLOR BASIC PROGRAMS & OTHER HELPFUL HINTS

Debugging Compiled Programs

If your CBASIC-3 program compiles without errors, but does not perform as expected, chances are you made a logical programming error. Make sure that the Program and Data storage areas do not overlap. If variable storage is allocated in the upper 32K of RAM, did you use the RAM64K page# statement? Make sure there are no DISK buffers (#1-#9 vars) in the upper 32K. Does the CBASIC-3 program overlap or conflict with another program being called?

The CBASIC-3 program listing provides you with some valuable information that can be used to find run-time errors in conjunction with the TRACE statement and a Monitor/Debugger program, like Cer-Comp's TRSMON System Monitor. The statement addresses on the listing can be used to set Breakpoints at the beginning of a specific program line. The Symbol table dump at the end of the listing shows variable memory locations, that can be examined with the Monitors memory examine and change function. With this information you can tell whether or not the program is running correctly up to the point where you examined the variables.

Read the Manual carefully, there is a great deal of information in this document which can make programming in CBASIC-3 easier for you. If CBASIC-3 is your first experience with a compiler, it would be wise to read this manual more than once. Many of the questions we get about programs are answered in the manual, so before you call or write to us with a question, check the manual, chances are the answer to your question is in here.

Errors During Compilation

When CBASIC-3 detects an error in the source program during compilation, the source line in error and a message describing the error will be displayed. The line immediately below the source line in error will have an "up-arrow" showing the approximate position of the error. Note that on long statement lines it may be more than one line below the last line to indicate that the error is on the second, third or fourth display line. This error locating arrow is about 95% accurate. When an error is detected the compiler will not process any further information on the line, even if it is a multiple statement line. So examine the rest of the line carefully for possible undetected errors. If an error should occur during compilation, "DO NOT" attempt to execute the compiled program, as the program is incomplete and undetermined results will occur. Maybe even crash or wipe out a disk.

DIFFERENCES BETWEEN CBASIC-3 AND COLOR BASIC PROGRAMS
&
OTHER HELPFUL HINTS

Converting Color Basic Programs
VARIABLE INITIALIZATION

Many Color Basic programs can be converted to CBASIC-3 compiled programs easily. However, in many cases the program will assume that all variable storage is cleared at run time. In CBASIC-3 this is not the case, variables are not initialized automatically. This can cause very strange results when the compiled program is executed. If variable initialization is required in a program, it can be done by assignment statements: A=0:B=0:A$="":etc. This can take a lot of program code depending on how many variables are to be initialized. Another method can be used that will produce less program code and clear all variables to a 0 or "" state. This method uses a FOR/NEXT loop and the VARPTR function. In the beginning of the program, use a variable name not used in the program and assign it a value of 0 , then follow it with a GOSUB to a line # past the end of the program. Example: YY=0:GOSUB 9990

For the last lines of the program write a FOR/NEXT loop using another previously unused variable name in the following form: FOR ZZ=VARPTR(YY) TO VARPTR(ZZ)-1. The final form of the initialization routines would look something like this:

```
10 YY=0:GOSUB 9990
.
9990 FOR ZZ=VARPTR(YY) TO VARPTR(ZZ)-1
9995 POKE ZZ,0:NEXT
```

This is one of the easiest and most effective ways to initialize variables in a CBASIC-3 program.

DIMENSION STATEMENTS & STRINGS

Another area of confusion when converting Color Basic programs is String Variable arrays. In Color Basic a Dimensioned String array only has one element in its definition: DIM A$(10). In Color Basic this means to allocate 10 different strings, A(1) thru A(10). In CBASIC-3 it means to allocate 1 string 10 characters in length. The reason for this is to allow better control over variable storage allocation and eliminate the problems associated with "String Pools" and "Garbage Collection" at run-time. When you define a string array in CBASIC-3, you must tell it the number of elements in the array and the length that is to be reserved for each string: DIM(10,32). This would allocate space for 10 strings of 32 characters each. This can be changed easily when converting a Color Basic program, by using the Editor in CBASIC-3 to search for DIM statements and then use the Line Edit function to change it to the correct format. CBASIC-3 will also automatically allocate space for an array up to 10 elements without requiring it to be declared in a DIM statement. However, remember that each element is assigned only 32 bytes of space the same as a default string variable.

DIFFERENCES BETWEEN CBASIC-3 AND COLOR BASIC PROGRAMS
&
OTHER HELPFUL HINTS

STRING VARIABLES

CBASIC-3 will normally allocate 32 bytes of storage for a string variable unless it is decaired in a DIM statement. In some Color Basic programs you may see an assignment statement in which the string being assigned is longer than 32 characters. In these cases you will have to use the DIM statement in CBASIC-3 to allocate enough space for the string variable or it will be truncated to 32 characters. This can cause an "FC" Function Call error in a program when the string is used as part of a DRAW or PLAY statement. Use the TRACE function to locate the line # that is causing the problem.

Example 1:

```
10 A$="BM10,10;C2;U8R6F2D2L8BR8D2G2L6BR12U8R8D8"
20 DRAW A$
```

Example 2:

```
5 DIM A$(50)
10 A$="BM10,10;C2;U8R6F2D2L8BR8D2G2L6BR12U8R8D8"
20 DRAW A$
```

The first example would cause an FC error when the program is run since the assignment would only move the first 32 characters to the variable. In the second example the variable A$ was first assigned 50 characters of space in the DIM statement before being assigned the string. It would execute correctly.

GRAPHICS GET & PUT ARRAYS

Most of the time Color Basic programs that use the Graphics GET & PUT statements will define an array large enough for CBASIC-3 to use. Sometimes a program will use a different method than the one mentioned in the Extended Color Basic manual to calculate the size of an array for a GET or PUT. In these cases the array may not be large enough for CBASIC-3 to use. This can produce a "FC" Function Call error at run-time. If you encounter this problem refer to the CBASIC-3 manual section on GET & PUT statements to check the dimension calculations. Also use the TRACE function to locate the line # causing the error.

CBASIC-3 LANGUAGE SUMMARY

ASSIGNMENT STATEMENTS:		LET	POKE	DPOKE	DATA	READ
RESTORE	LSET	RSET	SWITCH			
CONTROL STATEMENTS:		EXEC	CALL	RUN	FOR	NEXT
STEP	GOTO	GOSUB	RETURN	IF/THEN/ELSE	STOP	END
ON/GOTO	ON/GOSUB	ON ERROR GOTO	ON BRK GOTO	ON OVR GOTO	ON NOVR GOTO	ON RESET GOTO
STACK	CHAIN					
INTERRUPT CONTROL STATEMENTS:		ON KBDIRQ GOTO	ON TMRIRQ GOTO	ON SERIRQ GOTO	ON IRQ GOTO	ON NMI GOTO
ON FIRQ GOTO	ON SWI GOTO	RETI	IRQ on/off	IRQ = mask	SWI	IRQ simulate
FIRQ simulate	NMI simulate					
INPUT/OUTPUT STATEMENTS:		OPEN	INPUT	LINE INPUT	PRINT	PRINT @
WRITE	CLOSE	FIELD	GET	PUT	RESTORE	KILL
RENAME	DSKI$	DSKO$	VERIFY	DRIVE	CLOADM	LOADM
CSAVEM	SAVEM	GETCHAR	PUTCHAR	BRATE	PRATE	DSEARCH
AUDIO on/off	MOTOR on/off					
EXTENDED MEMORY STATEMENTS:		RAM64K page#	RAM on/off	LPEEK	LPOKE	DLPEEK
DLPOKE	LPCOPY					
HI-RES TEXT SCREEN STATEMENTS:		HIRES(8 modes)	WIDTH	LOCATE	ATTR	HSTATUS
COMPILER DIRECTIVES:		BASE	ORG	GEN	END	DIM
REM	DPSET	MODULE	PCLEAR	PAUSE on/off	CBLINK	UNLINK
NUMERIC FUNCTIONS:		ABS	POS	POS@	RND	PEEK
DPEEK	TAB	ASC	LEN	INSTR	VAL	ERR
EOF	SWAP	LOF	LOC	FREE	CVN	VARPTR
JOYSTK	BUTTON	INKEY	TIMER	OVEREM	SGN	ERL
INT						
STRING FUNCTIONS:		CHR$	LEFT$	RIGHT$	MID$	STR$
TRM$	STRING$	MKN$	INKEY$	BUF$	HEX$	SWITCH$
SWAP$						
SOUND & GRAPHICS STATEMENTS:		PLAY	SOUND	(H)CIRCLE	(H)COLOR	(H)CLS
(H)DRAW	(H)GET	(H)PUT	(H)LINE	(H)PAINT	PCLS	PCOPY
PMODE	PSET	PRESET	(H)RESET	(H)SCREEN	(H)SET	(H)POINT
PPOINT	HMODE	BORDER	HPRINT	HBUFF	PALETTE	RGB/CMP

ARITHMETIC OPERATORS	LOGICAL OPERATORS	RELATIONAL OPERATORS
+ ADD	& LOGICAL AND	<,>,= GREATER/LESS THAN, EQUAL
- SUBTRACT	! LOGICAL OR	<=, >= LESS/EQUAL, GREATER/EQUAL
/ DIVIDE	% LOGICAL XOR	<> NOT EQUAL
* MULTIPLY	# LOGICAL NOT	AND / OR
- NEGATE	+ CONCATENATE STRING	

CBASIC III

CBASIC-3 Run-Time ERROR CODES

One of the following codes will be generated if an error occurs during the execution of a compiled program. The ERR or ERNO function will return the most recent error generated, provided ON ERROR trapping is active. A determination can be made by the program, based on the error condition, to attempt correcting the error, abort the program or whatever the programmer decides.

If ON ERROR trapping is disabled, a normal Basic error message will be displayed and control will then be returned to Color Basic. At this point there are basically two options available. Either to press the Reset button to Cold Start the computer or re-execute the program with an EXEC statement. If for some reason the compiled program was corrupted, re-execution may cause the computer to crash or some other unpredictable results may occur. For this reason it is recommended that a POKE&H71,0 be performed and the Reset button pressed to insure that the computer is cleared to its normal state.

01 Next Without For, should not occur
02 Syntax error, cause unknown
03 Return Without, should not occur
04 Out of data in READ statement
05 Illegal function call, use TRACE to locate line#
06 Multiply overflow, results exceeded +32767 to -32768
07 Out of Memory, Illegal procedure call
08 Undefinde line, should not occur
09 Bad Subscript, should not occur
10 Attempt to Redimension an array, should not occur
11 Divide by zero attempted
12 Illegal Direct Statement, should not occur
13 Variable and data type mismatch
14 Out of String Space, should not occur
15 String too long, should not occur
16 String formula too complex, should not occur
17 Cannot continue, should not occur
18 Bad file data
19 File already open, disk or tape
20 Bad device number
21 Input or Output device error (hardware ?)
22 File Mode error, attempted input from output device, etc.
23 File not open for I/O
24 Attempted to input more data than a file contained.
25 Direct Statement, should not occur
26 Undefined function attempted
27 File does not exist (disk)
28 Bad random access disk record number.
29 Disk is full, no more room to write
31 Disk is write protected on attempted write
32 Bad file name
33 Disk file structure is corrupted.
39 Hires Graphics Error
40 Hires Print Error

CBASIC III
Hi-Resolution Text Package

The Hi Resolution Text Package is designed to improve the standard 32*16 and WIDTH 40/80 Screen displays. The program is fully integrated into the compiled Basic program by using the "HIRES" statement. It also allows you to switch back and forth between the Hi-Res format and the Standard 32 by 16 format for complete compatibility in almost all situations. The format of the display when the compiled program is first executed defaults to 80 characters by 24 lines. This can be changed to 32, 40, or 64 characters in either 192 or 225 Resolution modes thru the use of control codes. The package also includes other control code functions which add an extensive amount of flexability to the display. Some of them include: Reverse Screen, Reverse character, Underline character, Double Size characters, Erase to end of line, Erase to end of screen, Clear Screen, Home Cursor, Bell tone, and more. All of these features are controlled thru the use of control code characters sent via the CHR$(n) Basic statement or thru Machine language routines. The following is a list of Control codes recognized by the program and the function that it performs.

CHR$(n)	Function
1	Display Black characters on a White background (Default)
2	Display White characters on Black background.
6	Switch between Blinking & Non-Blinking Cursor
7	Sound Bell tone.
8	Backspace cursor one character position.
9	Advance cursor one character position.
10	Move cursor down one line (Scroll if at bottom).
11	Initiate X,Y cursor position function.
12	Clear screen.
13	Move cursor to begin of line & move down 1 line.
14	Turns character Underline off (default).
15	Turns character Underline on.
16	Home cursor to position #0 on the screen (top left).
17	Turns Destructive Cursor on.
18	Turns Destructive Cursor off (default).
19	Turns Space character Underline On (Default)
20	Turns Space character Underline Off
21	Erase from cursor to the end of line.
22	Erase from cursor to the end of screen.
23	Turn Reverse character mode off (Default).
24	Turn Reverse character mode on.
25	Save current cursor position.
26	Restore cursor to previously saved position.
27	Change chars/line, or Auto key repeat
28	Change display to Monochrome or Color mode.
29	Switch Screen format to Hi-Res or Standard 32*16.
30	Turns double size characters off (default).
31	Turns double size characters on.

Control Code Use

All of the screen control functions will be used with the Basic statement "PRINT CHR$(n)". For example, to clear the screen use the Basic statement PRINT CHR$(12). There are several control functions which are not completed with a single character code. The first one, (11), is used for X,Y cursor positioning and the second one , (27), has two functions depending on the value of the character immediately following it.

The X,Y cursor position function allows the cursor to be positioned to any location on the screen with a minimum of effort. This can be useful for screen mapping & information updating. This is similar to the Basic PRINT @ function. Instead of using a single number for the location, a column position and line number are used. These values must immediately follow the X,Y control code. A column value of 0 to the current number of characters per line may be used (51 is the default). The line number must then follow with a value from 0 to 23. For example, to position the cursor to the middle of the screen and print the word "HELP", you would use the following statement:

```
PRINT CHR$(11);CHR$(23);CHR$(11);"HELP"
```

This would print the word "HELP" starting at column 23 on line 11. Notice that a ";" must be used between each character so that other characters are not sent in between the column, line #, and print data for the command to work correctly.

"Escape" Character Sequence Commands

The "Escape" code CHR$(27) is used for three different functions depending upon the value of the character following it:

1) The number of lines on the Hi-Res Screen to be protected
2) The number of characters per line to be displayed on the Hi-Res Screen
3) Clearing several of the Special functions options with a single command

Changing Characters per line

The Hi-Res Screen package allows the user to set the number of characters displayed per line on the Hi-Res Text Screen. This can be varied from 32 to 80 characters per line in defined steps. The Hi-Res screen defaults to a 80 characters across by 24 lines in 225 Resolution at program startup time, but can be changed to one of 8 different formats. The following characters correspond to the number of display characters per line selected when used following the "Escape" code:

```
1 = 32 (192)        2 = 40 (192)
3 = 64 (192)        4 = 80 (192)
5 = 32 (225)        6 = 40 (225)
7 = 64 (225)        8 = 80 (225) default
```

```
PRINT CHR$(27);"5"<enter>  Set width to 32, 225 Res.
PRINT CHR$(27),"64<enter>  Set width to 64, 192 Res.
```

Clearing Special functions

There is a special function code used to reset most of the special functions in the HI-Res package. The functions which are reset to the default conditions are:Reverse Display (2), Underline (15), Reverse character mode (24), Double Size characters (31), Destructive Cursor (18) and Protected lines (27). All of these functions can be reset by the single command:

```
PRINT CHR$(27);"0"
```

This can be useful for clearing display options used during a program that has been interrupted while some of these functions were in use, or at the end of a program using them.

Changing Screen Formats

This function allows the user to switch screen formats back and forth between the Normal 32*16 screen and the Hi-Res screen. When in the Standard 32 by 16 screen all Hi-Res control functions will be ignored except the "CHR$(28)" which is used to return you back into the Hi-Res Screen format. This function toggles or flips back and forth between formats each time it is entered.

Changing Monochrome or Color modes

This command allows the user to select whether or not to supress the color display thru a single controle character. The screen comes up in Monochrome mode by default and be changed by sending a CHR$(28). Each time it is send, the screen flips between Mono & Color mode, the Basic command would be PRINT CHR$(28).

Character Highlighting Functions

The majority of the control functions supported consist of a single control code and can easily be used in a Basic program. Three of the functions control how the characters will be displayed on the screen until they are turned off. They are Underline CHR$(15), Reverse characters CHR$(24), and Double size characters CHR$(31). Once these functions are enabled, each character displayed will be affected by the active functions. Any combination of the three or all three may be enabled at the same time. They may also be reset at any time. The Reverse character effect can also be obtained by adding a value of 128 to any normal ASCII printable code. For example, to highlight a single character just add 128 to the letter using the format:

```
PRINT CHR$(ASC("Z")+128)
```

The Destructive cursor function allows you to tell the program whether or not to erase the character at the current cursor location. This is normally on by default. Some screen editing programs require it to be off to function correctly, while others require it to be on, so characters are erased during backspace operations. For these reasons we allow it to be changed.

The Reverse screen function also allows for special effects, or just personal preference for the screen display.

Additional Functions

Three functions have been added to allow more flexibility in using Hi-Res. The first allows you to switch between a blinking and non-blinking cursor CHR$(6). The second function allows you to select whether or not to Underline space characters on the screen CHR$(19) & CHR$(20) (spaces underlined by default). The third function allows you to select whether or not to erase the screen to the end of line following a carriage return or "Enter" character CHR$(25) & CHR$(26) (erase to end of line is off by default). All three of the additional functions consist of a single control code and remain in effect until turned off or switched by their counterpart code.

EFFECTS ON BASIC SCREEN COMMANDS

This package was designed to be as compatible and convenient to use as possible, so normal operations with CBASIC-3 programs would be affected as little as possible. Since some Basic programs use commands that affect the screen display, we have tried to make them as compatible as possible with the new screen format. Unfortunately, this may not be 100% compatible but should be close enough so the programs will still run without any major problems. If problems do arise, you can always switch back to the standard screen format for those functions, and then back to the Hi-Res format with a simple function command.

One of the most common screen commands is "CLS", the clear screen command. With the Hi Resolution package in the compiled program, this command only clears the screen in normal video (black characters on a white background). If a value follows the command, it will clear the standard 16*32 Text screen to that color.

The second most used screen command is the PRINT @ function. Under normal system operation this value may not exceed a value of 511, or an error will occur. You only have 32 character positions available per line by 16 lines, thus 0 to 511 is the range. When using the Hi Resolution screen package any value is allowed and will be adjusted according to the number of characters displayed per line. For example, if you printed at column 68 in the 40 character mode, it would display on line 2, column 28. If you did the same thing in 64 character mode, it would display on line 2, column 4. If you would like to have compatibility with the old screen format, just reprogram the number of characters per line to 32. This is accomplished by the statement:

```
PRINT CHR$(27);"1"<enter>
```

When in this mode, all PRINT @ screen formatting should be almost identical to the original format.

DISK DIRECTORY PROGRAM LISTING

```
0010 '  This is a demonstration program that shows
0020 ' how to use the BASE & DIM statements to map
0030 ' out an area of memory for reading and analyzing
0040 ' a disk directory using DSKI$.  Since the variable
0050 ' arrays NA$ and EX$ are setup to map out the DSKI$
0060 ' variables A$ & B$.  By using this method of re-mapping
0070 ' variables, extracting information is very fast
0080 ' since extensive string manipulation is not necessary.
0090 '  This method of variable mapping must be used any
0100 ' time the DSKI$ function is used to read mixed
0110 ' binary and ASCII information from disk.  The reason
0120 ' for this, is string functions use a 00 as an end
0130 ' of string marker for strings shorter than the
0140 ' defined length.  Thus when mixed ASCII & binary
0150 ' data are read using DSKI$, string functions will
0160 ' not allow access to any of the information in the
0170 ' string variable past the first 00 encountered.
0180 '
0190 BASE=$5000 : ' start variable space at $5000
0200 DIM A$(128),B$(128): 'variables for DSKI$
0210 BASE=$5000 : ' put next variable at same place in memory
0220 DIM NA$(7,32) : ' map directory names every 32 bytes 0-7
0230 BASE=$5008 : ' map Extensions at name + 8 for each entry
0240 DIM EX$(7,32) : ' map Extensions every 32 bytes 0-7
0250 BASE=0 : 'restore variables allocation to normal
0260 CLS:INPUT "DRIVE TO ANALYZE";D
0270 FOR S=3 TO 11: ' read sectors 3 thru 11 of directory track
0280 DSKI$ D,17,S,A$,B$ : ' read sector on track 17
0290 FOR L=0 TO 7:' loop for 8 entries per sectory
0300 IF PEEK(VARPTR(NA$(L)))=$FF THEN 330:' empty entry
0305 PRINT LEFT$(NA$(L),8);".";
0310 PRINT LEFT$(EX$(L),3),PEEK(VARPTR(NA$(L))+11),
0320 IF PEEK(VARPTR(NA$(L))+12)=0 THEN PRINT "B" ELSE PRINT "A"
0330 NEXT L,S: ' loop for 8 entries & all sectors
0340 INPUT A:GOTO 260
```

DISK COPY PROGRAM LISTING

```
0001 OPT N: '  Option for no listing generated
0002 ' This example program demonstrates how to use the
0003 ' DSKI$ & DISKO$ with a string array to copy the
0004 ' contents of a disk to another disk.  The destination
0005 ' disk must have been previously formatted.  It is
0006 ' equivalent to having a "BACKUP" command.
0007 '   The program uses a string array to store the entire
0008 ' contents of a disk track for each read/write sequence.
0009 '   You could use the BASE & DIM statements to put the
0010 ' track buffer array anywhere in memory that doesn't
0011 ' cause a conflict.
0012 '
0015 DIM A$(36,128):' Setup string ARRAY for Track buffer
0020 CLS:INPUT "DRIVE TO COPY FROM AND TO";CF,CT
0030 FOR T=0 TO 34: ' Loop for all 35 tracks
0040 FOR S=0 TO 34 STEP 2: ' loop for 1-18
0050 DSKI$CF,T,(S/2)+1,A$(S),A$(S+1): ' read sector into array
0060 NEXT S
0070 FOR S=0 TO 34 STEP 2: ' loop for sectors 1-18 write(track)
0080 DSKO$CT,T,(S/2)+1,A$(S),A$(S+1): ' write sector from array
0090 NEXT S,T:    ' Next Sector & Track
0100 INPUT "COPY COMPLETE, ANOTHER COPY Y/N";A$
0110 IF A$="N" THEN END ELSE RUN
```

DISK MENU PROGRAM LISTING

```
0010 ORG = $6000: HIRES:  ' Include Hires Text package
0013 PRINT CHR$(27);"5";:'  SCREEN MODE 32 CHARS/225 RES
0015 POKE $FFD9,0: ' HIGH SPEED
0020 BASE = $600 : DIM A$(128),B$(128): 'variables for DSKI$
0040 BASE = $600 : DIM NA$(7,32) : ' dir names every 32 bytes 0-7
0060 BASE = $608 : DIM EX$(7,32) : ' map .ext name+8  bytes 0-7
0080 BASE=0 : 'restore variables allocation to normal
0090 DIM FI$(68,12):  'array for all file names possible
0091 CLS
0092 PRINT "     MENU MASTER PROGRAM": PRINT
0100 PRINT "T - TEXTPRO3    C - CBASIC3"
0110 PRINT "E - EDTASM3     D - DPIII+"
0111 PRINT "M - MONIIIA     I - LASER "
0112 PRINT "S - SOURCE      Q - QUIT "
0113 PRINT "L - DISKLOOK    Q - QUIT "
0115 PRINT
0120 INPUT "DRIVE # OR COMMAND KEY ";C$
0125 IF C$="" THEN 091
0130 C= INSTR("TCEDMISLQ",C$)
0140 IF C=0 THEN D=VAL(C$):GOTO 200
0150 ON C GOTO 160,170,180,190,191,192,193,194,195
0155 GOTO 91
0160 CHAIN"TEXTPRO3.BIN:2"
0170 CHAIN"CBASIC3.BIN:2"
0180 CHAIN"EDTASM3.BIN:2"
0190 CHAIN"DPIII+.BIN:2"
0191 CHAIN"MONIIIA.BIN:2"
0192 CHAIN"LASER.BIN:2"
0193 CHAIN"SOURCE3.BIN:2"
0194 CHAIN"DISKLOOK:2"
0195 PRINT"EXITING PROGRAM BACK TO SYSTEM"
0196 POKE$71,0:POKE$FFD8,0:EXEC DPEEK($FFFE):END
0200 FOR I = 0 TO 68: FI$(I)="":NEXT: I = 1
0205 DRIVE D
0210 FOR S=3 TO 11: ' read sectors 3 thru 11 of directory track
0220 DSKI$ D,17,S,A$,B$ : ' read sector on track 17
0230 FOR L=0 TO 7:' loop for 8 entries per sectory
0240 PK=PEEK(VARPTR(NA$(L)))
0250 IF PK=$FF OR PK=0 THEN 290:' empty entry
0260 IF DPEEK(VARPTR(NA$(L))+11)<>$200 THEN 290 :' NOT BINARY
0270 FI$(I)= LEFT$(NA$(L),8)+"."+LEFT$(EX$(L),3)
0280 I = I + 1 :' NEXT ARRAY ENTRY
0290 NEXT L,S: ' loop for 8 entries & all sectors
0300 IF I = 1 THEN RUN :' NO EXECUTABLE FILES
0310 FOR F = 1 TO I - 1
0320 PRINT F;"-"; FI$(F),: ' display executable binary files
0330 NEXT: PRINT:PRINT
0340 INPUT"ENTER NUMBER OF FILE TO EXECUTE";F
0345 IF ((F>=I) OR (F<=0)) THEN 360
0350 PRINT "LOADING ";FI$(F):CHAIN FI$(F)
0360 RUN
```

GRAPHICS PRINT PROGRAM LISTING

```
0010 OPT S,N
0020 ORG =$6000
0030 MODULE
0040 BASE=$0600 : ' start variable space at $5000
0050 DIM A$(128),B$(128): 'variables for DSKI$
0060 BASE=$0600 : ' put next variable at same place in memory
0070 DIM NA$(7,32) : ' map directory names every 32 bytes 0-7
0080 BASE=$0608 : ' map Extensions at name + 8 for each entry
0090 DIM EX$(7,32) : ' map extensions every 32 bytes 0-7
0100 BASE=0 : 'restore variables allocation to normal
0110 DIM FI$(68,12)
0120 RAM64K $30
0130 ON ERROR GOTO 660
0135 MODE=3:FG=0:BG=63
0140 PRATE=9600
0145 '*****************************************
0146 '*     SET GRAPHICS MODE & DISPLAY       *
0147 '*****************************************
0150 STACK=$6000:HMODE MODE
0160 PALETTE 0,FG:PALETTE 1,BG:PALETTE 8,FG:PALETTE 9,BG
0170 BORDER BF:  TIMER=0:IRQ OFF
0180 IF TIMER<120 THEN 180 ELSE HMODE 0
0185 '*****************************************
0186 '*      MAIN MENU PROMPT & INPUT         *
0187 '*****************************************
0190 CLS:PRINT "pRINT, lOAD, sAVE, gRAPHICS"
0200 INPUT "mODE, dIRECTORY OR eXIT";A$
0210 IF A$="" THEN RAM64K 255:END
0220 A=INSTR("PLSGMD",A$):  IF A=0 THEN RAM64K 255 : END
0230 ON A GOSUB 250,430,520,150,710,790
0240 GOTO 140
0245 '*****************************************
0246 '*  LASER LINE6 GRAPHICS SCREEN DUMP     *
0247 '*****************************************
0250 INPUT "ENTER RESOLUTION 75, 100, 150, 300 ";RE$
0260 INPUT "DO FORM FEED WHEN DONE (Y/N)";FF$
0270 INPUT "START POSITION 0 OR 1 ";SP$
0280 HMODE MO
0290 MODE$=CHR$(27)+"*t"+RE$+"R"
0300 START$=CHR$(27)+"*r"+SP$+"A"
0305 IF(MODE=1)OR(MODE=3)THEN TW=80:TW$="080" GOTO 310
0306 TW=160:TW$="160"
0310 TRANSFER$=CHR$(27)+"*b"+TW$+"W"
0320 AD=$8000: 'GRAPHICS IN 8000-FF00
0330 IF SP$<>"0" THEN PRINT #-2,CHR$(10)
0340 PRINT #-2,MODE$;START$;:'SEND MODE & START CMDS
0350 FOR LINE = 0 TO 191
0360 FOR DEPTH = 1 TO 2
0370 PRINT #-2,TRANSFER$;:' SEND TRANSFER CMD
0380 FOR COL = 0 TO TW-1
0390 PUTCHAR #-2,PEEK(AD+COL):NEXT COL,DEPTH
```

```
0400 AD=AD+TW:NEXT LINE
0410 PRINT #-2,CHR$(27);"*rB";
0415 IF FF$="Y" THEN PRINT #-2,CHR$(12);
0420 RETURN
0425 '*******************************************
0430 '*           LOAD GRAPHICS PAGE            *
0431 '*******************************************
0440 INPUT "ENTER NAME & DRIVE TO LOAD";FI$
0450 OPEN"R",1,FI$,256
0460 HMODE MO
0465 Y =LOF(1):IF Y>30720 THEN Y=30720
0466 CLOSE #1:OPEN "I",1,FI$
0470 FOR X=$8000 TO X+Y STEP 1
0490 GETCHAR #1,A
0500 POKE X,A: NEXT X
0510 CLOSE #1:RETURN
0515 '*******************************************
0520 '*           SAVE GRAPHICS PAGE            *
0521 '*******************************************
0560 IF MODE AND 1 THEN Y =$BFFF ELSE Y = $F7FF
0590 INPUT "ENTER FILE NAME & DRIVE";FI$
0600 OPEN"O",#1,FI$
0610 FOR X=$8000 TO Y STEP 1
0620 PUTCHAR #1,PEEK(X)
0630 NEXT X
0640 CLOSE #1
0650 RETURN
0655 '*******************************************
0660 '*               ERROR HANDLER             *
0661 '*******************************************
0670 HMODE 0
0680 PRINT "ERROR #";ERR;" IN LINE #";ERL
0690 INPUT "PRESS ENTER TO RESTART";A$
0700 GOTO 120
0701 '*******************************************
0710 '*           SET GRAPHICS MODE             *
0711 '*******************************************
0720 HMODE MO
0730 GETCHAR #0,A
0740 IF A=8 THEN MODE = MODE -1
0750 IF A=9 THEN MODE = MODE +1
0760 IF MODE <0 THEN MODE=4
0770 IF MODE >4 THEN MODE= 0
0780 IF A=13 THEN RETURN ELSE 720
0781 '*******************************************
0790 '*           DIRECTORY DISPLAY             *
0791 '*******************************************
0800 CLS:INPUT "DIRECTORY DRIVE #";C$
0810 IF C$="" THEN RETURN
0820 D=VAL(C$)
0830 FOR I = 0 TO 68: FI$(I)="":NEXT: I = 1
0840 DRIVE D
0850 FOR S=3 TO 11: ' read sectors 3 thru 11 of directory track
```

```
0860 DSKI$ D,17,S,A$,B$ : ' read sector on track 17
0870 FOR L=0 TO 7:' loop for 8 entries per sectory
0880 PK=PEEK(VARPTR(NA$(L)))
0890 IF PK=$FF OR PK=0 THEN 920:' empty entry
0900 FI$(I)= LEFT$(NA$(L),8)+"."+LEFT$(EX$(L),3)
0910 I = I + 1 :' NEXT ARRAY ENTRY
0920 NEXT L,S: ' loop for 8 entries & all sectors
0930 IF I=1 THEN 970:' NO ENTRIES
0940 FOR F = 1 TO I - 1
0950 PRINT F;"-"; FI$(F),
0960 NEXT: PRINT
0970 INPUT"PRESS ENTER TO RETURN TO MENU";F$
0980 RETURN
```