CBASIC III

Color Computer 3 Basic Compiler

Bill Vergona

Table of Contents

CBASIC III

128/512K COC0-3 DISK EXTENDED Extended Color Basic COMPILER & program EDITING SYSTEM

Revision A

CER-COMP 5566 RICOCHET AVE. LAS VEGAS, NEVADA 89110

(702) 452-0632

LIMITED LICENSE TO USERS

Cer-Comp grants you, the owner and original purchaser of CBASIC III, a limited license for incorporating CBASIC III 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 wit hin 30 days of first publication.
3. The Author or publisher must pay a royalty 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 preceding 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 III 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 wit h 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.

Text Layout
Ruby Gem - ASCIIDOCTOR-PDF

Sources
available upon request

Editor
Pete Willard

Date
October 2019

In this revised version of the document, all references to fees and obtaining support are deprecated.

INTRODUCTION

CBASIC III 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 Processor 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 III is used to Create and/or Edit programs for the CBASIC III 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 over typing 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 III compiler is an optimizing 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 subroutines 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 III can produce programs which may reflect a 5 to 1000 times speed improvement over an interpreter. Since CBASIC III 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.

SCOPE AND REFERENCES

This manual is written to acquaint the user with the features of the CBASIC III 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 III 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 III 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.

Table 1. New Keyboard Characters

New Combination	Output	Value	Legacy Keys
CLEAR + 0	\	$5C	SHIFT + CLEAR
CLEAR + 1	|	$7C	*n/a
CLEAR + 2	~	$7E	*n/a
CLEAR + 3	[	$5B	SHIFT + ↓
CLEAR + 4	]	$5D	SHIFT + →
CLEAR + 5	^	$5E	↑
CLEAR + 6	_	$5F	SHIFT + ↑
CLEAR + 7	`	$60	*n/a
CLEAR + 8	{	$7B	*n/a
CLEAR + 9	}	$7D	*n/a

STARTUP PROCEDURES

CBASIC III 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 III 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 COC0-3 has 512K of memory installed, CBASIC III will automatically install 2 RAMDISK's as drives 2 & 3. These RAMDISK's 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 III as well as any of our disk programs.

TEXT EDITING COMMANDS

TEXT EDITOR

Conventions Used in the Manual

\

The "Backslash" is the character displayed when the SHIFT + @ keys are depressed as a delimiter for the **SEARCH** & **REPLACE** commands. Also see editor command summary.

()

When items are enclosed within these characters are required by that command to perform correctly.

[]

When items are enclosed within these characters are considered to be optional, when used they must also be in the required order.

< >

When items are enclosed within these characters they are comments.

ENTER

is used to denote an ENTER key press and is used to signify the completion of a line entry.

-

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

←

The Left arrow is recognized as a Backspace.

BREAK

This key is used to regain control at any time and will return to the **READY** prompt. If BREAK is depressed during a line entry or edit, any changes or recent 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. 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.

PRINTING

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 al low those users not having a printer to abort an accidental printer request and not hang up the system.

LIST

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 Extended Color Basic `LIST` function.

Example

```
LIST 100-300 ENTER
```

RENUMBER

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 lines exceed 9999 before the end of file, the increment is automatically decreased. The re-numbering is repeated until a workable value is reached.

Example

```
RENUMBER 5 100 ENTER
```

This will Re-sequence the line numbers in the file and increment each line number by '5', beginning with '100'.

DELETE

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

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 (RP)

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 then the entire file will be used. If only the starting line # is specified then the replace will be from the line # to the end of file. If both start & end line #'s are specified then only the lines including the range 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

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 [EDITOR KEYS] under [AEDIT].

Example

`LEDIT 110` ENTER

Edit line number 100 using the edit functions.

AUTO EDIT

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 of the file or a until 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 ↓ 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.

Editor Keys

Table 2. **EDIT FUNCTION KEYS**

FUNCTION	**DEPRESS**
MOVE CURSOR RIGHT	→
MOVE CURSOR RIGHT l WORD	CLEAR

FUNCTION	DEPRESS
MOVE CURSOR LEFT (backspace)	←
INSERT SINGLE SPACE	SHIFT + ↑
MULTIPLE CHARACTER INSERT on/off	SHIFT + @
DELETE CHARACTER	SHIFT + ↓
MOVE CURSOR TO END OF LINE	SHIFT + →
MOVE CURSOR TO BEGIN OF LINE	SHIFT + ←
GOTO NEXT SEQUENTIAL LINE	↓
GOTO PREVIOUS LINE	↑
END LINE AT CURSOR POSITION	SHIFT + CLEAR
REPLACE OLD LINE WITH NEW	ENTER
EXIT FROM EDIT MODE	BREAK

COPY

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 ENTER
```

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

MOVE

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

The **MOVE** command works almost exactly the same as the **COPY** command 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 ENTER
```

This would move the lines from 1100 thru 1345 to the next line following line 100.

AUTOMATIC LINE NUMBERING

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

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

SIZE

MEMORY SIZE COMMAND

Syntax: `SIZE` ENTER

Example
Responds with:

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

PRINTER

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 III` command, `PR CBASIC III`

Example

`PRINTER NLINE LIST` ENTER

This would tell the editor to list the file to the printer with no line numbers.

EXIT

Syntax : `EXIT` ENTER

Causes control to return to `BASIC`*. Once you exit CBASIC III you cannot return or re-execute the program, it must be re-loaded from disk*

NEW

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.

BRATE

PRINTER BAUD RATE

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 III` 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:

Value	Baud Rate
0	110
1	300
2	600
3	1200
4	2400
5	4800
6	9600

Example: `BRATE 5` ENTER
Set baud rate to 4800 baud

LF

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.

RD (Key Repeat)

AUTOMATIC KEY REPEAT DELAY COMMAND

Syntax : `RD` < 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 del ay 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*

SW (Screen Width)

SCREEN WIDTH - Set 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.

Value	Width	Value	Width
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 t he display. The 225 mode gives an extra pixel width between lines so that the descender 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 :
`SCREEN 63 0` ENTER
BUff chars/Black Background
`SCREEN 18 0` ENTER + *Green chars/Black Background*

CC

CHANGE COLOR/MONOCHROME MODE

Syntax : `CC` ENTER

This command allows the user to force the computer to suppress 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

TEXT EDITOR I/O COMMANDS

SAVE

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
```

LOAD

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 accessible for editing. When a duplicate output file is encountered it is automatically removed by the Extended Color Basic 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.

APPEND

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.

Example

```
APPEND BIOIA:3
```
Open the file BIOIA on drive #3 for input and append it into the available text buffer.

DRIVE

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
```

DIR

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

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 Extended Color 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

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 useful for testing for syntax errors or for 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`*.*

THE CBASIC III program STRUCTURE

A CBASIC III program consists of a series of "source lines". A source l ine consists of a line number followed by one or more CBASIC III 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 Extended Color Basic program which has line numbers greater than 9999, renumber the program using Extended Color Basic before saving the program to disk in ASCII format.

Spaces in CBASIC III statements are not required, however they may be used to improve readability (except when used in string constants or following variable names that precede 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 FOR I=1 TO N: 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 III program is very similar to that of the Extended Color Basic interpreter. Most of the CBASIC III statements are identical in format to Extended Color Basic. Many programs may be written with the interpreter for testing and debugging, then saved to disk in ASCII format, loaded into CBASIC III and compiled. Most of the syntax differences between CBASIC III and Extended Color Basic can be used in the interpreter for testing and debugging. However, there are some syntax formats in Extended Color Basic that cannot be used in CBASIC III.

These minor differences will become apparent as you use CBASIC III, and should not pose much of a problem in converting existing Color Basic programs.

ARITHMETIC OPERATIONS

NUMBERS

CBASIC III'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 III 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 III 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	Normal Positive number
-5000	Negative Number
$100	Hexadecimal notation
&H1000	Hexadecimal Notation
#1	Inverse of 1
#$5000	Inverse of Hexadecimal 5000

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
-1 $FFFF #0 65635 &HFFFFF	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 these number constants have the same binary value.

NUMERIC VARIABLES

Legal numeric variable names in CBASIC III 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`	Signle Letter Lower Case
`N`	Single Letter Capitalized
`xx`	Letters Lower Case
`ZX`	Letters Capitalized
`r2`	Letter + #, Lower Case
`A0`	Letter + #, Capitalized

`ZIP`	More than 2 letters, only 1st two are used. IE; `ZI`)

If declared 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 III does not perform run-time subscript error checking for over range 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. Obviously, 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 subscripts

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

CBASIC III 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 equivalent 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 number (UNARY)

There are also four boolean operators:

& or AND	Logical AND
| or OR	Logical OR
%	Exclusive OR
#	Logical NOT aka Complement or Invert (UNARY)

For those who see UNARY and say "huh?" It means, a operation that does not require 2 or more parts. "2+2=4" is not a unary operation because there are 2 parts and a result while "-2" is a unary operation because it makes a value "negative".

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 III 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 III may produce machine instructions which are shortest and fastest. Expressions are evaluated in the following order:

- Numeric Functions
- Unary Negative and Not
- Multiplication, Division
- Addition, Subtraction
- Relational tests `<`,`>`,`=`, ,`>=`,`<>`
- 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)
$200+ZX + A&BIC*D/F+(H+(J*2)&$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). The absolute value of a real number is the number of units from 0 the number is. The absolute value of 7 for example is 7 becasue it is that many units away from zero. The absolute value of -7 however is 7. So it in a way gets rid of the negative.

Example

```
abs(5) is 5
abs(-5) is 5
```

INT(expr)

Converts the numeric expression to an integer (For Extended Color Basic testing)

RND(expr)

Returns a random integer between l 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 (+l), negative (-1) or zero (0).

TIMER

Returns the contents or allows setting the timer 0-65535. 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 immediately 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 III representation. CBASIC III 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: REM ADD L.S. BYTES
150 A1=A1+1   : REM ADD 1 TO MS BYTES FOR CARRY
200 A1=A1+B1  : REM ADD MS BYTES
```

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

```
100 A2=A2-B2 : ON OVR GOTO 200 : REM SUB. LS BYTES
150 A1=A1- B1 : RETURN : REM 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 III 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 a single 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 to process decimal numbers as strings can also be used to allow a wider range of decimal values to be input and processed.

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(l). 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 : REM DEFINE ARRAY, CLEAR TOTAL
110 FOR I= 1 TO 10 :REM 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 : REM ADD THE CENTS TOGETHER
240 TO=V(0,I) +T0+T1/100 :REM ADD THE DOLLARS & CENTS OVER 100
250 T1=OVEREM: REM CENTS = REMAINDER 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$(T0)
330 ' CONVERT CENTS TO STRING B$ WITH DECIMAL POINT
340 ' ADD A LEADING ZERO IF ITS VALUE IS LESS THAN 10
350 B$=". "+RIGHT$("00"+STR$(T1),2):REM  ONLY 2 DECIMAL PLACES
360 PRINT A$;B$ : GOTO 100 :REM 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 III variable names so the CBASIC III program may easily manipulate them.

3) Use CBASIC III'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.

BASIC Functions

Hardware Functions

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 toa number between 0 and 3 to be valid. The following values will select the different Joystick buttons:

0	Right Button 1	Old joystick (Single Button)
1	Right Button 2	Newer Joystick (Dual Button)
2	Left Button 1	Old joystick (Single Button)
3	Left Button 2	Newer Joystick (Dual Button)

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 t he 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)
```

Special Functions

Run-Time SWITCH variables

Syntax: `SWITCHn` = *numeric expression*

Example

```
var = SWITCHn
SWITCH$ = string expression
var$ = SWITCH$
```

The `SWITCH` variables are run time variables that occupy the first 16 bytes ($00-$0F) on the direct page of memory used by the CBASIC III program. There are eight numeric variable switches that can be uses or one 16 byte string variable, `SWITCH0-SWITCH7` and `SWITCH$`.

They both occupy the same memory space and can be used like any other variables in CBASIC III. They can be useful for temporary variable storage or for passing variables to & from machine language programs or subroutines and CBASIC III programs.

Since the `SWITCH` variables are not initialized by the CBASIC III program, they can be useful for passing information to a CBASIC III program before it is executed or from one CBASIC III program to another "CHAIN-ed" CBASIC III program that uses the same Direct Page of memory.

Example

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

STRING Processing

CBASIC III features a complete set of string processing capabilities which allow CBASIC III programs to perform operations on character oriented data. Character type data is represented in CBASIC III 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 III 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. There are various allowed formats for using string arrays that require only one subscript (which may also be an expression).

Some Examples

```
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.

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 III 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$`.

Example

`LEFT$("Example",3)` returns "Exa"

MID$(X$,N,M)

returns a string which is that part of the string `X$` beginning wit h its Nth character and extending for *M* characters.

Example

`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$.

Example

`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.

Example

`STR$(1234)` returns the "string" "1234". This is the opposite 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.

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.

Example

`TRM$("Example ")` returns "Example" + Removes the space after "e"

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.

With regards to 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.

Example

`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.

Examples

`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.

Example

`VAL("123")` returns the numeric value of `123`. `VAL("THREE")` results in an error.

String Operations on the I/O Buffer

Commonly the BASIC language has limitations because of the input formatting when reading mixed data types. For example; BASIC input conventions cause commas that are part of the input data to break up what is a really long string, etc. CBASIC III 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.

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$`:

Example

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

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 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 buff er expansion if operations on extremely complex string expressions are involved.

Examples

```
"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.

Compiler Directives

ORG, BASE and DPSET

Syntax: `ORG = address`

`BASE = address`

`DPSET = address(MSB only)`

These statements are used to control how CBASIC III 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 III 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 III 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.

Example

```
ORG=$4000
```

This will cause instructions generated for any following BASIC statements to begin at address $4000. The ORG statement may be used to create "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 III 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 immediately following the compiled program, and the BASE address pointer is initialized to 0000, which means that it is not being used currently.

CBASIC III 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 III 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 us i ng only the BASE pointer for variable allocation.

When using the ORG and BASE statements the programmer must take care to e nsure 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.

Examples

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

PCLEAR

Syntax: `PCLEAR` *[0-8]*

The PCLEAR statement is normally associated with Graphics. You use it to clear (reserve) a number of graphics pages. In CBASIC III 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 III 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

Syntax: `DIM` *variable* `(dimension)`

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.

Examples

```
DIM R(65) DIM W(8), X(8) `
`DIM AR$(8,25)
```

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.

Example

```
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 III 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 III 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.

`CT=N`	To write the RS232 command/ctrl registers
`N=DS`	To read the RS232 data & status registers
`N=XY`	Read X & Y location regs on the X-PAD

REMARK Statement

Syntax: `REM` or `{rem}` *followed by comment text*

The `REM` statement is used to insert comments in the BASIC source program. The first three letters must be `REM` or the first character is a single quote `{rem}`. All characters following the `REM` or single quote charadter are considered to be comments until an end of line or until a colon `:` character is reached.

The `REM` statement does not affect the object program size or speed as it does not generate any code.

Examples

```
REM This is a comment
{rem} ALSO A COMMENT
```

TRACE

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 Extended 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

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

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 tha t are to be passed from the calling compiled program must be coordinated by the BASE and DIM or SWITCH statements if required.

Example

```
MODULE
```

I/O Structure Changes

CBASIC III 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 Extended 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 LOAD`M & `EXEC commands to execute another CBASIC III 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 Lherefore provided two additional commands to allow you to manually Link or Unlink the CBASIC III program.

UNLINK

This command will "unlink" or restore the original vector information to be the same as it was found before the program was executed (normal Extended Color Basic vectors). It would normally be used before you use the `CALL` or `EXEC` statements to exit from a CBASIC III program. When a program is unlinked, `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: REM unlink & do basic reset
```

CBLINK

```
This command will allow you to relink the {cb} program manually if you have previously unlinked it and
executed another program and returned. If the program has not previously been unlinked it will not try to
relink itself, so no conflict will occur.

_Example:_
```

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

Assignment Statements

LET (numeric)

Syntax: `LET` *(variable = expression **or** variable=expression)*

Used for arithmetic assignments. The expression is evaulated and the result is stored in the variable.

Use of the keyword LET is optional.

LET (string)

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.

Use of the keyword LET is optional.

POKE & DPOKE

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*.

DATA

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.

Example

```
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

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 1:

```
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
```

Example 2:

```
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

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 first, then the RESTORE statement resets the beginning of data again. The array B(l0) is same values from the DATA statement.

EXEC

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 Extended Color Basic ROM call compatibility. Upon returning from the EXECuted program or subroutine, the DP register will be restored from the stack automatically. 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 III program, use the BASE & DIM or SWITCH variable statements to coordinate returned values.

Example

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

CALL

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
```

Program Control Statements

FOR/NEXT

Syntax: `FOR` *var = expr* `TO` *expr* `STEP` *expr*
`NEXT` *(var),...,(var)*

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 III differs slightly from Extended Color Basic due to a looping method that results in extremely fast execution and minimum length.

The following are the characteristics of the `FOR/NEXT` operation:

- *var* must be a non-subscripted numeric variable.
- The loop will be executed at least once reguardless of the terminating value.
- 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.
- `FOR/NEXT` loops may be exited and entered at will.
- At compile time, up to 16 loops may be active and all must be properly nested.
- 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).
- 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

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

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

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 automat ically 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

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 III only checks for an ON BRK condition when data is being output to the screen.

Example

ON BRK GOTO 1000

ON-GOTO/ON-GOSUB

Syntax: ON *expr* `GOTO` $\langle$ *line#* $\rangle$,... ,$\langle$ *line#* $\rangle$ON *expr* `GOSUB` $\langle$ *line#* $\rangle$,... ,$\langle$ *line#* $\rangle$

The expression is evaluated and one line number in the list corresponding to the value is selected for a branch or subroutine call.

For Example

if $\langle$ *expr* $\rangle$ evaluates to 5, the 5th line number is used. If $\langle$ *expr* $\rangle$ 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

Syntax: STOP or END

The `STOP` and `END` statements are used to terminate execution of a program by causing a **Cold Start** return to the Extended 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

*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 f iles, like Extended Color Basic. It simply performs a `GOTO` to the first execution address of the program.

System Control Statements

GEN

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.

CLEAR

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

ON RESET GOTO

Syntax: `ON RESET GOTO` *line#*

This statement allows a CBASIC III 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 Extended Color Basic. The `ON RESET` statement is typically one of the first statements in a CBASIC III 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 pr oper code to re-initialize the **Direct Page** and **Stack** registers and 64K RAM if used.

Example

```
ON RESET GOTO 5000
```

STACK

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
```

The stack builds downward

PAUSE

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 III only). Normally `PAUSE` is enabled by default when a CBASIC III 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 III 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 III only checks for the Break key when data is being output to the Screen.

Example: `PAUSE OFF`

SIGN

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 III supresses this leading space so that multiple numeric variables can be output together to represent larger numbers. Since Extended Color Basic normally outputs this leading space, many programs expect it to be there when doing number to string conversions, etc. To make CBASIC III output the extra space, use the `SIGN ON` command.

Example

`SIGN ON` *enable leading sign space*

Interrupt Processing Statements

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 a t tempt 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 totall y invalid or even hang the system.

ON INTERRUPT

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 even 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 O 8 keys you would poke a "01" at **$FF02**. Also, at any time an `INKEY`, `INPUT` or other command that causes a keyboard scan (`PRINT` with `PAUSE` enabled) will change the value of **$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= l 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.

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 III 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 III program, as you actually don't.

A few points to remember are that **ALL** interrupt handling subroutines must end with a `RETI` statement or you will have a crashed system. If you wish to disable one of the interrupt handlers that have already been in use, then use the same statement without a *line#*.

Example

`ON KBDIRQ GOTO` instead of `ON KBDIRQ GOTO 1000`

IRQ

Syntax: `IRQ =` *value*
`IRQ ON`
`IRQ 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.

The `IRQ=` statement is used to selectively enable or disable **GIME** interrupt conditions . There are six different interrupt conditions that can be enabled by this stat ement 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 in location **$FF92 IRQ** or **$FF92 FIRQ**

If you wanted to enable the **Keyboard** and **Timer** interrupts you would use a value of 34 (2 for the `KBD` + 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.

Example:

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

These are **GIME** interrupts and require that location **$FF90** be setup to allow interrupts to reach the CPU. Examples: `$FC` for COCO2
`$7C` for COCO 3 Screens.

Other ON INTERRUPT Statements

Syntax: `ON FIRQ GOTO` *line#*Must Save Registers Manually* `FIRQ`
`ON NM! GOTO` *line#* `GEN $3406` to **PSHSD** at start of `IRQ` code + `ON SW! GOTO` *line#* `GEN $3506` to **PULSD** before `RETI`

These statements are used for generating programs where interrupts are processed by specific service routines rather than by the normal Extended 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 MCC6809 interrupt structure works.

RETI

INTERRUPT RETURN

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

IRQ, NMI, FIRQ, SWI

Syntax : `IRQ` : `NM!` : `FIRQ` : `SWI`

These commands allow you to simulate an interrupt via software in a CBASIC III 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` and `SWI3` interrupt vectors are reserved for use by the compiled programs and are not available for use by the programmer.

Extended Memory Management

LPOKE and DLPOKE

Syntax: `LPOKE` *page#, offset, value*+ `DLPOKE` *page#, offset, value*

`LPOKE` behaves differently than Extended Color Basic

`DLPOKE` is not available in Extended Color 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 while `DLPOKE` will store the full 16 bit value.

Examples: `DLPOKE 6,0,255`

`LPOKE 6,0,&HFFFF` `LPOKE P,OF,VA`

LPEEK and DLPEEK

Syntax: `A=LPEEK(`page#,offset`)`
`A=DLPEEK(` *page#,offset* `)`

`LPEEK` behaves differently than Extended Color Basic
`DLPEEK` is not available in Extended Color Basic

The `LPEEK` and `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,0F)
```

RAM64K

Syntax: `RAM64K` *page#*

`RAM64K` is not available in Extended Color Basic

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.

Examples

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

The preceeding examples demonstrate 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 III 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 III program.

You can still use the upper portion of memory $8000-$FDFF for variable storage etc, but with a twist. You must tell CBASIC III what the starting page# is for 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, `RAM64K 60`.

Examples

`RAM64K 48`

`RAM64K $30`

`RAM64K 255` **Note: This last one does nothing useful.**

RAM ON/OFF

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 opposite, 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

Syntax : `LPCOPY` *source* TO *destination*

Not available in Extended Color Basic

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 source and destination pages.

Hi-Res Text Screen Statements

WIDTH

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

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)`

ATTR

Syntax : `ATTR` *foreground, background, Blink, Underline*

The `ATTR` or **Attributes** command allows you to select the foreground (Character) and background colors for the `WIDTH 40` and `WIDTH 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 foreground/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
ATTR F,B,B
```

- 1st example will underline
- 2nd example will blink