



BasTo6809

Color Computer Basic Compiler

Glen Hewlett

Table of Contents

BasTo6809 User Manual	1
Introduction	1
Version Information	1
Dependencies	1
Compiling BASIC Programs	1
Command Line Options	1
Special Features	4
Changes to BASIC's Graphic features	5
New Commands Or Features Added To Basic	7
How to use BasTo6809	8
Preparing Your System To Compile Basic Programs	8
Optimizing	9
How to Install	11
Command Line Compilation	12
Compile the Integrated Development Environment - IDE	15
Compiling Programs for 64k RAM CoCo's	17
Compiling for a Dragon computer	18
Supported BASIC commands	19
Color Basic Commands	19
Numeric Commands	22
String Commands	24
Logical operators	25
Math operators	25
GETJOYD Technical Details	25
Statement Reference List	26
SDC Related Commands	30
SDC Commands related to SD card directories	32
SDC Commands for Audio Playback	33
Floating Point Commands	34
Floating Point String conversion commands:	35
Floating Point Comparison commands:	35
Additional Floating Point Details	37
Get Command	43
Graphics commands:	44
Sprite Handling - Overview	47
CoCo 1 & 2 Graphic Modes	48
COCO 3 Graphic Modes	48
Limitations of the Compiler	51
Error Handling	52
Thanks	53
Change Log	54

BasTo6809 User Manual

Introduction

BasTo6809 is a compiler that converts a BASIC program into 6809 Assembly Language, designed to run on the TRS-80 Color Computer. The assembly code generated by BasTo6809 is ready for use with `lwasm` from the LWTTOOLS^[1] collection, allowing you to assemble and execute machine language programs on your CoCo.

This tool is ideal for anyone looking to take their BASIC programs and convert them to a 6809 assembly language for faster execution or to speed up development of assembly language code.

Version Information

BasTo6809

Version: 4.10

Author

Glen Hewlett

GitHub

BASIC-To-6809 <https://github.com/nowhereman999/BASIC-To-6809>

Dependencies

- **QB64pe** (Phoenix Edition) <https://www.qb64phoenix.com/>
- **LWASM** <http://www.lwtools.ca/>

Optional

FFMPEG <https://www.ffmpeg.org/>

Compiling BASIC Programs

Usage

```
BasTo6809 [options] program.bas
```

Where **program.bas** is the BASIC program you wish to convert to 6809 assembly language.

By default, the compiler will output a fully commented assembly language file (**program.asm**) that can be processed by **LWASM** to produce a machine code program for the Color Computer.

Command Line Options

BasTo6809 provides several command-line options to customize the behaviour of the compiler:

-coco

Use this option if your input is a tokenized Color Computer BASIC program.

-ascii

Use this option for a plain text BASIC program written in ASCII format, such as a program created with

a text editor or QB64.

-bx

Optimizes the branch lengths, affecting how efficiently LWASM assembles the program.

```
-b0 (default): Some branches may be longer than necessary, resulting in larger/slower programs.  
-b1 Ensures all branches are as short as possible, producing smaller & faster programs, but will slow  
down the assembly process.
```

-a

Makes the program autostart after it is loaded. Displays the version number of BasTo6809.

-ox

Controls the optimization level during the compilation process:

```
-o0 disables optimizations (not recommended).  
-o1 enables basic optimization.  
-o2 (default) enables full optimization for the fastest and smallest possible code.
```

-pxxxx

Specifies the starting memory location for the program in hexadecimal. Useful if you need some extra space reserved for your own program. The default starting location for the compiled program is \$2600 which is the memory space after the first **PMODE** graphics screen.

Example: **-p4000** sets the starting address at \$4000.

-sxxx

This option sets the maximum length to reserve for strings in an array. The default (and maximum) is 255 bytes. If your program use smaller strings, setting this value can reduce the amount of RAM your program uses.

Example: **-s128** reserves 128 bytes for each string.

-fxxxx

Where xxxx is the font name used for printing to the **PMODE** 4 screen (default is Arcade). Look in folder Basic_Includes/Graphic_Screen_Fonts to see font names available

-Vx

Sets the verbosity level of the compiler output.

```
-v0 (default) produces no output during compilation.  
-v1 shows basic information while compiling.  
-vx x=2,3 or 4 more info is displayed while compiling
```

-k

Keeps miscellaneous files generated during the compilation process. By default, these files are

deleted, leaving only the .asm file.

-h

Displays a help message with information on how to use **BasTo6809**.



Locate will check to see if the location value exceeds that last available line and will adjust it to the last available line if it does.

[1] <http://www.lwtools.ca/>

Special Features

- You can write the program on a CoCo or on a modern computer using any text Editor
- New **GMODE** command allows you to choose every graphic mode the CoCo can produce, including semi graphics and if using a CoCo 3 all of the CoCo 3 graphics modes and Colour modes. Using these new screens you can use **LINE** (with B & BF), **CIRCLE** and **PAINT** commands.
- Use of line numbers is optional
- You can use Labels for sections of code to jump to (case sensitive)
- Variable names can be 25 characters long (case sensitive)
- Doesn't use any ROM calls so it's possible to use all of the 64k of RAM
- The assembly code generated is fully commented showing each BASIC line and how it is compiled. The assembly file generated can be used to help someone learn how to program in assembly language. Or allow an experienced assembly programmer to optimize the program by hand.
- Many new SDC related commands will allow you to read and write directly to the SDC filesystem from your BASIC program
- A new SDC audio playback command to play RAW audio samples directly from the SD card in the CoCoSDC
- Easily add assembly code anywhere you want in your program and easily share values of variables between BASIC and your assembly code.

Changes to BASIC's Graphic features

PMODE has been replaced by the **GMODE** command **PCLS** has been replaced by the **GCLS** command **PCOPY** has been replaced with the **GCOPY** command **LINE** command format has been changed to include a colour value and no longer uses **PSET** and **PRESET**.

The commands **PSET**, **HSET**, **PRESET**, **HRESET**, **PPOINT** or **HPOINT** are not supported. Instead they are replaced with **SET** and **POINT** commands. The compiler will use whichever graphic mode is set using the **GMODE** command and will **SET** pixels to the requested colour the user wants that matches the **GMODE** requested.

You can now use **SET,POINT,LINE,CIRCLE** & **PAINT** commands on every screen, even the regular text screen, using **GMODE 0,1**

GMODE - ModeNumber,GraphicsPage

Selects the graphics screen and the graphics page. ModeNumber is the graphics mode you want to use GraphicsPage is the Page you want to show/use for your graphics commands To see a list of ModeNumbers and the resolutions go here



Special note the ModeNumber must be an actual number and cannot be a variable as the compiler needs to know exactly which graphic mode commands to be included at compile time. GraphicsPage can be a variable. If you are going to use Graphic pages, the compiler needs to know how many pages to reserve in RAM (for CoCo 1 & 2 graphics). So you must have a "GMODE #,MaxPages" entry at the beginning of your BASIC program. Where the value of MaxPages will be an actual number and not a variable.

GCLS #

Colour the graphics screen

- # is the colour value you want the screen to be coloured

GCOPY SourcePage,DestinationPage

Makes a copy the Source graphics page to the Destination graphics page.

SourcePage

- Source graphics page

DestinationPage

- Destination graphics page

SET(x,y,Colour)

Sets a pixel on the screen

- x,y * Screen location of the pixel to be drawn
- Colour * Colour Number of the pixel to be drawn

POINT(x,y)

Returns the colour value of the pixel selected

- x,y * Screen location of the pixel value requested LINE(x0,y0)-(x1,y1),Colour[,B][F]
- x0,y0 * Starting location x1,y1 * Ending location
- Colour * Colour of the Line or Box to draw B * Draw a Box
- F * Fill the Box

PAINT(x,y), OldColour, FillColour

Fills the old colour value with the fill colour value which must also be the border colour of the section you are painting

- x,y - Starting location
- OldColour - Colour Number
- FillColour - Colour Number

CIRCLE(x,y), Radius, Colour

Draws a circle on the screen

- x,y - Origin of the circle
- Radius - Size of the circle, to keep the aspect ratio close to round Some of the graphics modes use scaling so the Radius isn't always a count of actual pixel values.
- Colour - Colour Number of the circle to draw

PALETTE v, Colour

Sets the CoCo 3 Palette value v -palette slot of 0 to 15

- v - Paletter Slot Number 0 to 15
- Colour - Colour value of 0 to 63

NTSC_FONTCOLOURS(b, f)

Background, Foreground - Sets the CoCo 3 NTSC font colours



DRAW, GET & PUT commands are not yet available

New Commands Or Features Added To Basic

- **IF/THEN/ELSE/ELSEIF/ENDIF**
- **SELECT/CASE**
- **WHILE/WEND**
- **DO/WHILE/LOOP**
- **DO/LOOP/UNTIL**
- **PRINT #-3**, Command allows you to print text on the **PMODE** 4 screen Use **LOCATE x,y** command to locate where on the **PMODE** screen to print the text.
- **PUT** Command can use all the usual options like **PSET** (default if no option is used), **PRESET**, **AND**, **OR**, **NOT** now can use **XOR** which XORs the bits on screen with the bits of the **GET** buffer as: **99 PUT(c,y)-(c+4,y+9),Sprite1,XOR**
- **SDCPLAY** Command that plays an audio sample or song directly off the SD card in the SDC Controller.
- **SDCPLAYORC90L**, **SDCPLAYORC90R**, **SDCPLAYORC90S** these commands are similar to **SDCPLAY** except the audio is sent to the Orchestra 90 or COCOFLASH cartridge.
- SDC file access commands that allow you to Read & Write files directly on the SD card's own filesystem.
- Floating Point commands (special commands to handle floating point calculations and operations)
- **GETJOYD** * Quickly get the joystick values of 0,31,63 of both joysticks both horizontally and vertically
- **SDC_LOADM** * Loads a file from the SD card into memory.
- **SDC_SAVEM** * Saves a file from memory to the SD card.
- You can include *inline assembly* code in your program using the **ADDASSEM** and **ENDASSEM** command to wrap the code in a special block of BASIC code. It will be included in the final assembly code and can be used to perform any special assembly language operations you need.

How to use BasTo6809

The compiler is called BasTo6809 and is itself written in BASIC, specifically QB64pe (Phoenix Edition). You'll need to compile BasTo6809 code yourself it using QB64pe to create the new executable files for your system. You will be instructing QB64pe to compile the basic programs into **EXE** files which can be run on your system.

To compile the programs, you'll need to download and install QB64pe from <https://www.qb64phoenix.com/>



QB64 PE is a modern extended BASIC+OpenGL language that retains QuickBASIC 4.5 and QBasic compatibility and compiles native binaries for Windows (7 and up), Linux, and macOS (Catalina and up).

After compiling BasTo6809 with QB64pe, make sure to include a folder named **Basic_Includes** and **Basic_Commands** alongside it. These additional folders contains **.asm** libraries and other includes, which the compiler will insert into the **output.asm** file it creates from your source **.BAS** file, if needed..

The last thing you will need to do is install **lwasm** on your computer, as this assembler is required to turn the compiler's assembly output into the final machine language program. <http://www.lwtools.ca/>



LWASM is part of LWTOOLS, which is a set of cross-development tools for the Motorola 6809 and Hitachi 6309 microprocessors.

Once you're compiling folder is prepared, it's fast and easy to compile your BASIC program to machine language using the following commands:

Using MacOS or Linux:

```
./BasTo6809 -ascii BASIC.bas.  
lwasm -9b1 -p cd -o./ML.bin BASIC.asm > ./Assembly_Listing.txt`
```

Using Windows:

```
.\BasTo6809 -ascii BASIC.bas.  
lwasm -9b1 -p cd -o./ML.bin BASIC.asm > ./Assembly_Listing.txt
```

At this point you'll have an *Executable* program called **ML.bin** in the folder that you can use on a real CoCo or an emulator.

Preparing Your System To Compile Basic Programs

1 Install and test QB64pe so you're familiar with how it compiles BASIC program

2 Install LWASM on your system

Once those two programs are installed on your system, you will use QB64PE to compile at least 3 files;

- *BasTo6809.bas*
- *BasTo6809.1.Tokenizer.bas*

- *BasTo6809.2.Compile.bas*



You should also compile cc1sl.bas so it's available, if needed.

After completing these steps, you're nearly ready.

Make sure your working directory now has the BasTo6809 executable, BasTo6809.Tokenizer executable, BasTo6809.Compile executable, and the cc1sl executable program (if you compiled it), as well as the sub folders named Basic_Includes and Basic_Commands, which has the supporting **.asm**, font files and other required files in them.

With that all setup you're now good to go, here's an example of how to compile a basic program called **HELLO.BAS** from the command line:

From MacOS or Linux:

```
./BasTo6809 HELLO.BAS
```

From Windows:

```
.\BasTo6809.exe HELLO.BAS
```

If the compiler doesn't report any errors then you should now have a file saved in your directory as **HELLO.asm**. You can look through the **.asm** file with any text editor to see the assembly code the compiler created. It generates an assembly language file with a lot of comments.

The next step is to use lwasm to assemble the program into a CoCo executable program, something like this:

```
lwasm -9b1 -p cd -o./HELLO.BIN HELLO.asm > ./NEW_Assembly_Listing.txt
```

This will create an output file called **HELLO.BIN** that you can now take and use on a real CoCo or an emulator and execute.

Optimizing

To generate the fastest and smallest version of your program use this command options below:



lwasm will take a while to assemble so be patient, could be a minute or so and it may seem like nothing is happening

For MacOS and Linux:

```
./BasTo6809 -b1 HELLO.BAS
```

For Windows:

```
.\BasTo6809 -b1 HELLO.BAS
```

The only other thing you might need to do if you have a program that is very big is use the **cc1sl** program. The steps for compiling a big program are:

For MacOS and Linux:

```
./BasTo6809 -b1 HELLO.BAS  
lwasm -9b1 -p cd -o./HELLO.BIN HELLO.asm > ./NEW_Assembly_Listing.txt  
./cc1sl -l HELLO.BIN -oBIGFILE.BIN
```

For Windows:

```
.\BasTo6809 -b1 HELLO.BAS  
lwasm -9b1 -p cd -o./HELLO.BIN HELLO.asm > ./NEW_Assembly_Listing.txt  
.\cc1sl -l HELLO.BIN -oBIGFILE.BIN
```

In this case your final program to execute on the CoCo is called **BIGFILE.BIN**, you can of course call it whatever you want. Remember to only use **cc1sl** if your file is fairly big.



I remember testing it with small programs and it seemed to not work. I never did look into why at least as of yet. But it works perfect if you do have a large program.

The latest version of the compiler can be found on my GitHub site. For support, ask for help on the CoCo Nation basic-to-6809 Discord channel.

GitHub:: BASIC-To-6809 <https://github.com/nowhereman999/BASIC-To-6809>

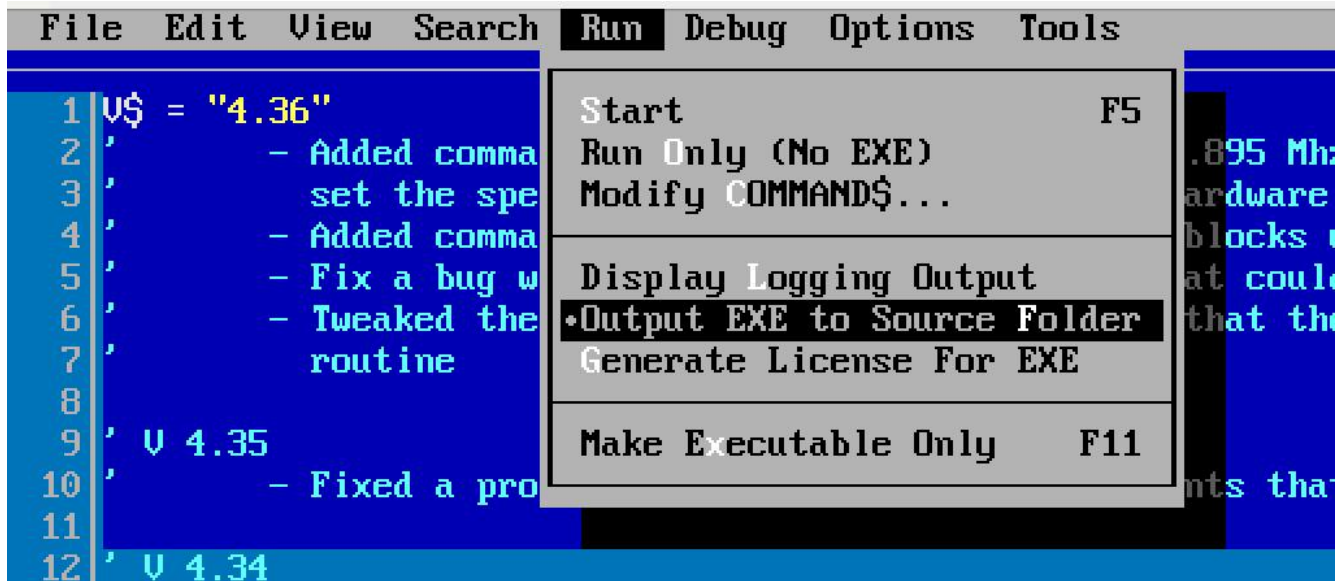
For support, ask for help on the CoCo Nation "basic-to-6809" Discord channel

How to Install

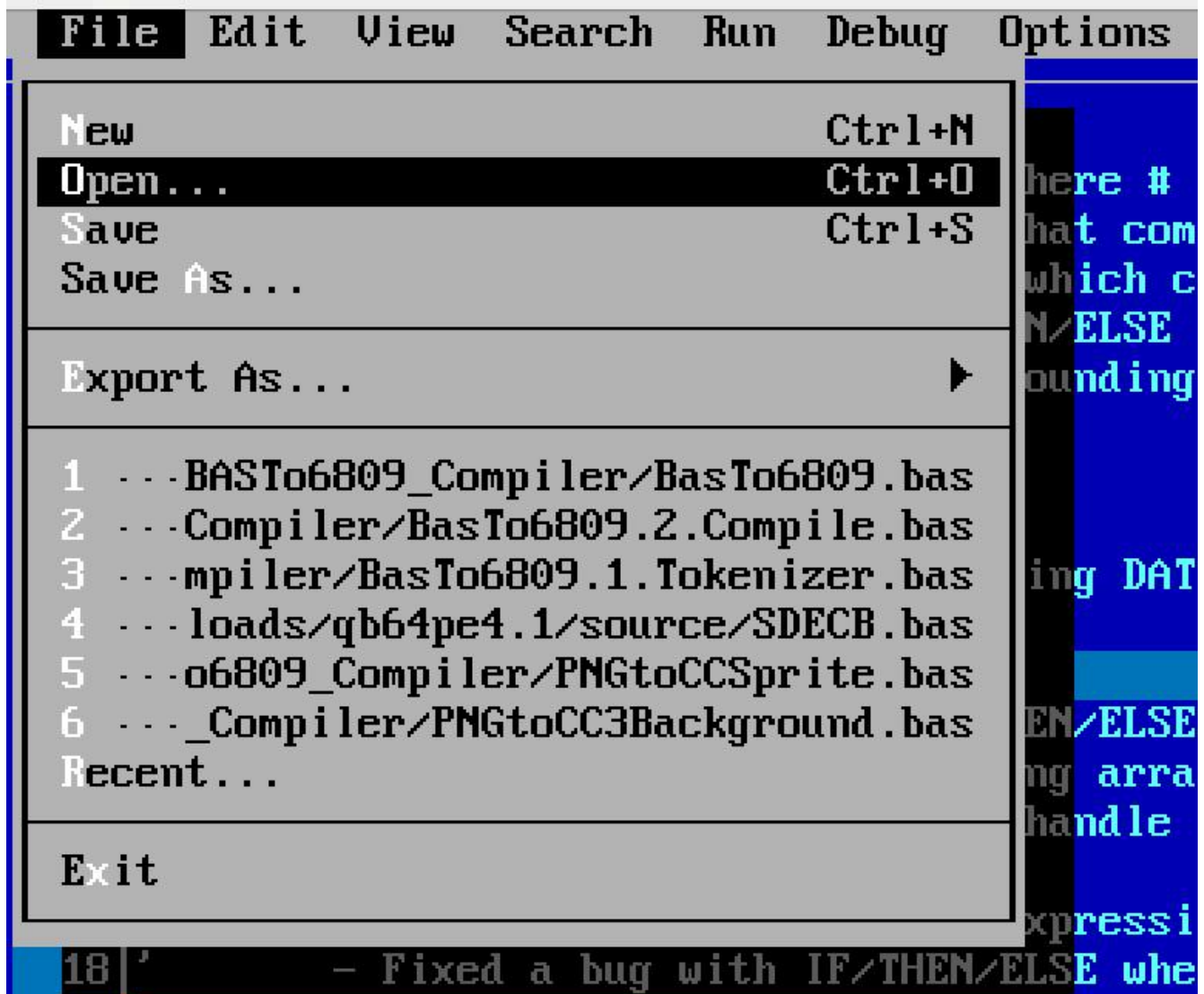
The compiler is called **BasTo6809** and is written in BASIC specifically QB64pe (QB64 Phoenix Edition). QB64pe is multi-platform so this BASIC to 6809 compiler can be used on a Mac, Linux or Windows machine.

Using QB64pe click on the Run menu option then click option "Output EXE to Source Folder"

There should now be a dot on the left of "Output EXE to Source Folder". This means this option is selected and the files that are compiled will be in the same folder as the source .bas files.



Next you will need to load the **.bas** files for the compiler. Use **File > Open** or **Ctrl-O** to open the file.



- Find the file **BasTo6809.bas** and choose/load it. Then press **F11** which will compile it and save the executable version in the same folder it was opened/loaded from.
- Open the file **BasTo6809.1.Tokenizer.bas** and once loaded press **F11**, again this will compile it and save the executable version in the same folder it was opened/loaded from.
- Open the file **BasTo6809.2.Compile.bas** and once loaded press **F11**, again this will compile it and save the executable version in the same folder it was opened/loaded from. (This will take awhile to compile)
- Open the file **cclsl.bas** and once loaded press **F11**, again this will compile it and save the executable version in the same folder it was opened/loaded from.

Command Line Compilation

When you have installed QB64PE on your system you can also compile the source files to executables using the command line. For convenience sake, it would necessitate that you add the location of QB64PE to you PATH ENVIRONMENT variable for you respective system.

When QB64PE is available in your path you can compile the source files using the following commands, for example:

```
QB64PE -x BasTo6809.bas
QB64PE -x BasTo6809.1.Tokenizer.bas`
QB64PE -x BasTo6809.2.Compile.bas
QB64PE -x cc1sl.bas
```

It could even be scriptable using a batch file or shell script or be added to a makefile for automated builds when new versions of the compiler are released.

Example Makefile (tweak as needed, assumes QB64PE is in your PATH):

```
ifndef $(OS),Windows_NT
OS := $(shell uname)
endif

ifndef OS
$(error "OS must be set to 'Linux', 'Windows_NT', or 'Osx'")
endif

ifeq ($(OS),Linux)

    CP := cp -r
    MKDIR := mkdir -p
    RM := rm -fr
    MV := mv
    SLASH := /

endif

ifeq ($(OS),Windows_NT)

    CP := xcopy /E /C /H /R /y
    MKDIR := mkdir
    RM := del
    MV := move /-y
    SLASH := "\"
endif

QB64PE = qb64pe
SRCS   = BasTo6809.1.Tokenizer.bas \
        BasTo6809.2.Compile.bas \
        BasTo6809.bas \
        cc1sl.bas

OUTS   = build/BasTo6809.1.Tokenizer.exe \
        build/BasTo6809.2.Compile.exe \
        build/BasTo6809.exe \
        build/cc1sl.exe
```

info:

```
@echo "This makefile will compile the BASIC-to-6809 source files"
@echo "into the executable files and move the results to the"
@echo "program folder."
@echo " "
@echo "Supply the operating system being used. lnx, win, osx"
@echo " "
```

all: \$(OUTS)

build:

```
$(MKDIR) build
```

build/%.exe: %.bas | build

```
$(QB64PE) -x $< && $(MV) $(@F) .$(SLASH)build$(SLASH)
```

clean:

```
$(RM) .$(SLASH)build$(SLASH)*.exe
```



This file assumes that MAKE is installed on your system. Windows users can install make from the Chocolatey package manager. Also, it does not have a handler for MacOS.

Compile the Integrated Development Environment - IDE

The IDE's name is **SDECB** (Super Duper Extended Color Basic) Similar to the other .bas programs you have just compiled you can now also compile the IDE. Once again File | Open this time go into the IDE folder and select/load **SDECB.bas**, once loaded press **F11** to compile it. This may also take awhile. Once finished you will have the executable version in the IDE folder.

For Mac and Linux users the executable files that have just been created won't have extensions. For Windows users the executable files that have just been created will have the extension .exe

Now that you have all the files compiled into executable files you can copy them all to a new working folder of your choice. From the IDE folder you must copy the executable file **SDECB.exe** and also the folder "internal" to your new working folder.

- If you're using Windows machine you should also copy the file "compile.bat" from this folder to your working folder.
- If you're using Mac or Linux machines you should also copy the file called "makefile" from this folder to your working folder.



The **compile.bat** and **makefile** are used by the IDE to compile your BASIC programs and gives you flexibility on what to do once you compile your BASIC program. These files can be edited so you can add functionality to your development environment.

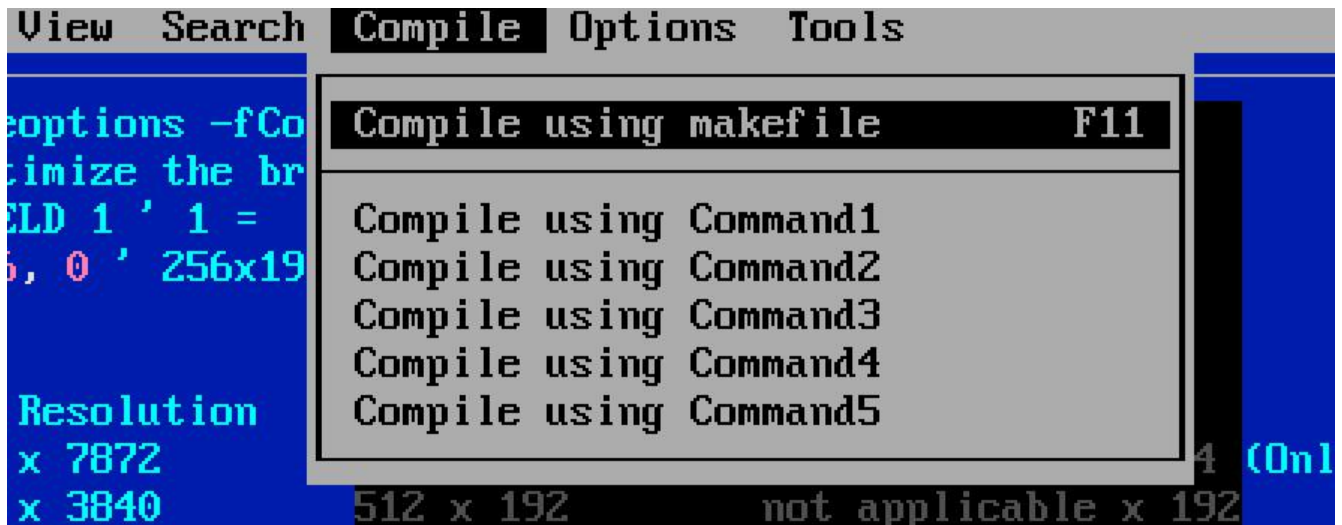
From the main folder (one folder up from the IDE folder) you need to copy the executable files, **BasTo6809.exe**, **BasTo6809.1.Tokenizer.exe**, **BasTo6809.2.Compile.exe** and **cc1s1.exe** (again on Mac and linux there won't be any extensions on the executable filenames) to your new working folder

You also must copy two folders "Basic_Includes" & "Basic_Commands" you may wish to copy the "Sample_Programs" folder if you want but it isn't necessary.

One last thing you will need to do is install **lwasm** on your computer as this is the assembler that is needed to turn the assembly output from the compiler into the final machine language program.

Make sure **lwasm** is either installed in your working folder or it is in your path. The **compile.bat** and **makefile** do expect **lwasm** to be in the working folder.

Once your working folder is setup you can use the IDE called **SDECB.exe** to write your BASIC program. The IDE will give you warnings if there are any commands it doesn't understand or there is a syntax error in your code that you can find and fix before you even try to compile your program. Once you're ready to compile your program you can Press **F11** or goto **Compile** and select the command you want to follow.



These Commands run commands in your compile.bat/makefile which you can edit to suite your needs. For example you could have Command5 compile the program, assemble it with **lwasm** then copy it to a floppy disk image and start your emulator of choice. If you look at the compile.bat or makefile you should be able to follow along the structure of the file to figure out how to add actions to the various command

You can also compile your BASIC programs to machine language from the command line using the following commands:

Using MacOS or Linux:

```
./BasTo6809 -ascii BASIC.bas
lwasm -9b1 -p cd -o./ML.bin BASIC.asm > ./Assembly_Listing.txt
```

Using Windows:

```
.\BasTo6809.exe -ascii BASIC.bas
lwasm -9b1 -p cd -o./ML.bin BASIC.asm > ./Assembly_Listing.txt
```

At this point you'll have an executable program called **ML.bin** in the folder that you can use on a real CoCo or an emulator.

Compiling Programs for 64k RAM CoCo's

If your program requires more than 32k you must use the **cc1sl** program (CoCo 1 Super Loader). This program enables the loading of an ML program no matter where it will be loaded into RAM including where the BASIC ROM addresses are.

cc1sl

CoCo 1 Super Loader v1.03 by Glen Hewlett

Usage: **cc1sl [-l] [-vx] FILENAME.BIN -oOUTNAME.BIN**

[-scn] or [-csv]

Turns a CoCo 1 Machine Language program into a loadable program no matter if it over writes BASIC ROM locations and more

Where:

-l

Will add the word LOADING at the bottom of the screen while the program loads

-vx

Amount of info to display while generating the new file x can be 0, 1 or 2. Default x=0 where no info is shown

FILENAME.BIN

is the name of your big CoCo 1 program, it must end with .BIN

OUTNAME.BIN

is the name of the output file to be created otherwise it defaults to **GO.BIN**



*.scn A binary file that must end with .scn will be shown on the CoCo text screen while loading *.csv A csv text file that must end with .csv will be shown on the CoCo text screen while loading



For more info see the **cc1sl_help.txt** file



cc1sl.bas is also a QB64pe program that you must compile with QB64pe before using.

Compiling for a Dragon computer

There are a few things you need to do in order to compile and run a program on a Dragon computer. Since the Dragon computer is very similar to a CoCo 1 code from the compiler that works on a CoCo 1 & 2 will pretty much execute as is on a Dragon computer. But the keyboard layout is different between the two computers. When compiling your program to target a Dragon computer you must use the `-dragon` option. This tells the compiler to include the proper keyboard routines that will work for the Dragon computer.

Also since the Dragon uses a different file format for loading binary files you must copy the **DRAGLOAD.BIN** file to your floppy disk or emulator disk image. You will also need to rename your compiled program to **COMPILED.BIN** and copy **COMPILED.BIN** to the same floppy disk or emulator disk image as **DRAGLOAD.BIN**.



You can rename "DRAGLOAD.BIN"

The steps you would take to compile a program called **HELLO.BAS**

- Compile your program into .asm file

```
./BasTo6809 -ascii -dragon HELLO.BAS
```

- Assemble the program into an DECB machine language binary file

```
lwasm -9b -p cd -o./HELLO.bin HELLO.asm
```

- Copy the DRAGLOAD.BIN file to a floppy disk or disk image and rename the file as **HELLO.BIN**

```
imgtool put coco_jvc_dgndos ./DISK1.DSK ./DRAGLOAD.BIN HELLO.BIN
```

- Copy your actual program HELLO.bin to a floppy disk or disk image, and rename the file as **COMPILED.BIN**

```
imgtool put coco_jvc_dgndos ./DISK1.DSK ./HELLO.bin COMPILED.BIN
```

Your disk should now have two files on it called **HELLO.BIN** (which is really the **DRAGLOAD.BIN** program) and **COMPILED.BIN** (which is your actual compiled program). The loader expects the file to be loaded from drive 1 and you start your program on the Dragon computer by typing:

```
RUN"HELLO.BIN" 
```

The **DRAGLOAD.BIN** program will handle programs that are 64k

Supported BASIC commands

The following is the current list of BASIC commands that are supported by the BasTo6809 compiler.

Color Basic Commands

The following is a list of BASIC commands that are supported by the compiler.

Word	Comments
'	This is the SHORTHAND for REM (remark)
AUDIO	ON/OFF - Connects or disconnects cassette output to TV speaker AUDIO ON or AUDIO OFF
BACK	
BACKUP	Not implemented yet
BSOUND	
CASE	
CIRCLE	x,y,r
CLEAR	Only clears all the variables to zero
CLOAD	Not implemented yet
CLOSE	Not implemented yet
CLS	
COLOR	
CONT	Not implemented yet
CONT	
COPY	Not implemented yet
CSAVE	Not implemented yet
DATA	
DEF	
DEL	
DIM	
DIM	
DIR	Not implemented
DLOAD	Not implemented
DO	WHILE/UNTIL See DO WHILE [LOOP]
DRAW	
DRIVE	Not implemented
DSKINI	Not implemented
EDIT	Not implemented yet
ELSE	

ELSEIF	
END	
ERASE	
EVERYCASE	
EXEC	
EXIT	DO/FOR/WHILE
FIELD	Not implemented
FILES	Not implemented
FOR	
FOR	
GCLS	
GCOPY	
GET	
GETJOYD	See technical data below
GMODE	
GOSUB	
GOTO	
IF	
INPUT	
IS	Not implemented
KILL	Not implemented
LET	
LINE	
LIST	Not implemented yet
LLIST	Not implemented yet
LOAD	Not implemented
LOADM	
LOCATE	
LOOP	See DO WHILE [LOOP]
LSET	Not implemented yet
MERGE	Not implemented
MOTOR	ON/OFF
NEW	
COPY	Not implemented yet
NEXT	
NTSC_FONTCOLOURS	Background, Foreground -Only used for GMODE 160 to 165. This sets the background and foreground colours of the NTSC composite output fonts.

OFF	
ON	
OPEN	Not implemented yet
PAINT	
PALETTE	
PCLEAR	Not implemented
PCOPY	Not implemented
PLAY	Used to play sounds
PLAYFIELD	# - Tells the compiler which playfield (scrolling) mode to use
PMODE	Use GMODE instead
POKE	
PRINT	Can't do PRINT USING, Use PRINT #-3,"Hello" to print on PMODE 4 screen
PUT	
READ	
REM	
RENAME	Not implemented yet
RENUM	Not implemented
RESTORE	
RETURN	
RSET	Not implemented yet
RUN	Not implemented yet
SAVE	Not implemented yet
SCREEN	
SDC_CLOSE	
SDC_DIRPAGE	
SDC_GETBYTE0	
SDC_GETBYTE1	
SDC_LOADM	
SDC_OPEN	
SDC_PLAY	* Multiple Variations. See SDC Commands for Audio Playback
SDC_PLAYORCL	
SDC_PLAYORCR	
SDC_PLAYORCS	
SDC_PUTBYTE0	
SDC_PUTBYTE1	

SDC_SAVEM	"FILENAME.BIN",#,Start,End,Exec
SDC_SETDIR	Expects a String
SDC_SETPOS	(#,a,b,c,d)
SELECT	SELECT CASE/END SELECT
SET	
SHOW	
SKIPF	Not implemented yet
SLEEP	
SOUND	
SPRITE	
SPRITE_LOAD	
STEP	
STOP	
THEN	
TIMER	
TO	
TROFF	Not implemented yet
TRON	Not implemented yet
UNLOAD	Not implemented yet
UNTIL	
USING	Not implemented
VBL	
VERIFY	Not implemented yet
WAIT	Example WAIT VBL
WEND	See DO WHILE [LOOP]
WHILE	See DO WHILE [LOOP]
WPOKE	16-bit POKE command
WRITE	Not implemented yet

Numeric Commands

Word	Comments
ABS	Absolute value of a number
ASC	ASCII value of a character
ATN	Arc Tangent of a number - Not implemented
BUTTON	Returns the current state of the joystick buttons
CMPEQ	Floating Point Compare if Greater Than

CMPGE	Floating Point Compare if Greater Than or Equal
CMPGT	Floating Point Compare if Greater Than
CMPLE	Floating Point Compare if Less Than or Equal
CMPLT	Floating Point Compare if Less Than
CMPNE	Floating Point Compare if Not Equal
COS	Conversion of angle to cosine - Not implemented
CVN	Not implemented yet
EOF	End-Of-File Not implemented
EXP	Exponential function of a number - Not implemented
FIX	Fixed-point value of a number - Not implemented
FLOATADD	
FLOATATAN	
FLOATCOS	
FLOATDIV	
FLOATEXP	
FLOATLOG	
FLOATMUL	
FLOATSIN	
FLOATSQR	
FLOATSUB	
FLOATTAN	
FLOATTOSTR	
FN	
FREE	Not implemented yet
IGNORE	
INSTR	
INT	
JOYSTK	
LEN	
LOC	Not implemented yet
LOF	Not implemented yet
LOG	Not implemented
MEM	Not implemented
PEEK	
POINT	
POINT_SG24	

POINT_SG6	
POS	Not implemented
POINT	
RND	Fast random number generator, will generate a value between 1 and x, where x can have a max value of 255
RNDL	Large random number generator, will generate a value between 1 and x, where x can have a max value of 32767
RNDZ	Fast random number generator, will generate a value between 0 and x, where x can have a max value of 255
SDC_DELETE	
SDC_GETBYTE	
SDC_INITDIR	
SDC_MKDIR	
SDC_SETDIR	
SGN	
SIN	Not implemented
SQR	
STRTOFLOAT	Convert a string to a Floating Point variable
TAB	
TAN	Not implemented
USR	Not implemented
VAL	
VARPTR	Not implemented
WPEEK	16-Bit PEEK command

String Commands

Word	Comments
CHR\$	
DSKI\$	Not implemented yet
DSKO\$	Not implemented yet
HEX\$	
INKEY\$	
LEFT\$	
LTRIM\$	
MID\$	
MKN\$	Not implemented yet

RIGHT\$	
RTRIM\$	
SDC_FILEINFO\$	
SDC_GETCURDIR\$	
STR\$	Not implemented yet
STRING\$	
TRIM\$	

Logical operators

AND
OR
XOR
NOT

Math operators

BASIC Command	Details	Comment
* +, -, *, /, ^,	The general math operators	
MOD	Remainder of division	
DIVR	Same as / except the result is rounded to the nearest value.	For compatibility DIVR accepts \ as integer division (which is the same as /)

GETJOYD Technical Details

This command will get the joystick values of 0,31,63 of both joysticks both horizontally and vertically. Results are stored same place BASIC normally has the Joystick readings:

Joystick	Vertical	Horizontal
LEFT	\$15A	\$15B
RIGHT	\$15C	\$15D

Statement Reference List

AUDIO - ON/OFF

Connects or disconnects cassette output to TV speaker **AUDIO ON** or **AUDIO OFF**

BUTTON()

Returns the current state of the joystick buttons **BUTTON()** See [GETJOYD Technical Details](#) Technical Details

CIRCLE

Draws a circle on the screen **CIRCLE x,y,radius** (Arcs are currently not supported)

CLEAR

ONLY Clears all variables to zero

CLS -c

Clears the screen **CLS c** where **c** is the color to use (default is 0=black)

0	Black
1	Green
2	Yellow
3	Blue
4	Red
5	Buff
6	Cyan
7	Magenta
8	Orange

CASE

SEE [\[SELECT\]](#) statement

DATA

Stores data in your program. Use **READ** to assign this data to variables. **DATA 5,3,BLOCKS**

DEF FN

Defines a user-defined numeric function **DEF FN (x) = x*3**

DIM

Declares variables **DIM a,b,c**

DO

SEE [\[LOOP\]](#) statement

DRAW

Draws a line on the screen **DRAW 'commands'** - **This command is not available yet.**



DRAW"BM100,100IS10IU25IBR25IND25IXA\$I"

ELSE

SEE **IF** statement

ELSEIF

SEE **IF** statement

END

Ends a program or a loop **END**

END IF

SEE **IF** statement

END SELECT

SEE [\[SELECT\]](#) statement

EVERYCASE

SEE [\[SELECT\]](#) statement

EXEC

Executes a program in memory **EXEC** or **EXEC 'start address'**

EXIT

Exits a program or a loop **EXIT**

DO/FOR/WHILE

SEE [\[LOOP\]](#) statement

FOR/NEXT - start - to - end

The FOR statement creates a counter loop using specified start and stop numerical boundaries. The default increment is + 1.

```
FOR _counter_ = _start value_ TO _stop value_ [step/increment]
```

Example:

```
FOR X = 1 TO 100
  Y = X * 10
NEXT X
```



Startvalue can be a literal or a variable, STEP/INCREMENT is 1 by default. NEXT will END the FOR loop when the STOP VALUE is reached.

GCLS -

Clears the graphics screen **GCLS** or **GCLS #** where # is the color to use (default is 0=black). This is now differentiated from legacy **PCLS** command.

GMODE - ModeNumber,GraphicsPage

Selects the graphics screen and the graphics page. ModeNumber is the graphics mode you want to use GraphicsPage is the Page you want to show/use for your graphics commands To see a list of ModeNumbers and the resolutions go here



Special note the ModeNumber must be an actual number and cannot be a variable as the compiler needs to know exactly which graphic mode commands to be included at compile time. GraphicsPage can be a variable. If you are going to use Graphic pages, the compiler needs to know how many pages to reserve in RAM (for CoCo 1 & 2 graphics). So you must have a "GMODE #,MaxPages" entry at the beginning of your BASIC program. Where the value of MaxPages will be an actual number and not a variable.

GCOPY - SourcePage,DestinationPage

Makes a copy the Source graphics page to the Destination graphics page.

GET

TBD

SELECT

Runs one of several groups of statements, depending on the value of an expression.

Example 1:

Run though the cases until a match is found:

```
SELECT CASE _test_value_
  CASE IS _value_comparison_
    [ STATEMENTS ]
  CASE _x_ TO _y_ [VALUE RANGE COMPARISON]
    [ STATEMENTS ]
END SELECT
```

Example 2:

Execute every case that matches the value of the expression:

```
SELECT EVERYCASE _number_
  CASE 1
    PRINT "Number is 1"
  CASE 2
    PRINT "Number is 2"
  CASE 3 to 5
```

```
PRINT "Number is between 3 and 5"
END SELECT
```

LOOP -[statements]

Repeats a block of statements while a Boolean condition is True or until the condition becomes True.

```
Do _WHILE_ | _UNTIL_ _condition_
{code}
Loop
```

or

```
DO
{code}
LOOP _WHILE_ | _UNTIL_ _condition_
```

Example:

```
' Poll INKEY until it has some content

[#LOOP]
DO
A$ = INKEY$
LOOP UNTIL A$ <> ""
```

PLAY

Plays music of specified note (A-G or 1-12), octave (O), volume (V), note-length (L), tempo (T), pause (P), and allows execution of substrings. Also sharps (# or +)and flats (-) .

```
PLAY"L1;A#;P8;V10;T3;L2;B-;9;XA$;"
```

PLAYFIELD

Scrollable Playfield Mode Selection

The PLAYFIELD # command sets the scrollable playfield mode in your BASIC program. The # must be a literal number, not a variable.

To create playfield graphics, use the command-line tool **PNGtoCC3Playfield** to convert a PNG into a file (or files). These can then be loaded into memory using one of the following commands in BASIC:

- **LOADM**
- **SDC_LOADM**
- **SDC_BIGLOADM**

Once loaded, the background playfield becomes scrollable according to the selected mode.

#	Max Resolution	Min Resolution	Size Multiple
1	256 x 7872	256 x 192	Not Applicable x 64
2	512 x 3840	512 x 192	Not Appliablte x 192
3	1024 x 512	1024 x 256	Not Applicable x 256
4	2048 x 256	512 x 256	256 x not applicable
5	2560 x 192	512 x 192	256 x not applicable

VIEW x,y

Command to select where in the scrollable playfield you want to view. The x & y co-ordinates are the top left corner of a viewable window of the playfield.

SDC Related Commands

Details about the new commands for users of the CoCoSDC.



You can use the new SDC_LOADM and SDC_SAVEM commands to directly read and write files and folders on the SD card installed in your CoCoSDC.

SDC_LOADM"FILENAME.BIN",#[,Offset]

Loads a machine language binary file into the computer from the SDC directly.

* is the file number 0 or 1
Offset * is optional, if it's included this amount will be added to the original LOADM address.

SDC_SAVEM"FILENAME.BIN",#,Start,End,Exec

Saves a section of memory to the SDC directly

is the file number 0 or 1
Start Address in memory to start copying from
End Address in memory to stop copying from
Exec Address where the program should start execution
Saves a section of memory to the SDC directly

SDC_OPEN"FILENAME.EXT",'X',#

Opens file for Reading from or Writing to the SD card directly.

FILENAME.EXT * can be any 8 character filename with a 3 character extension
X * is either an R for Read or W for Write
* is the file number to open. This must be either a 0 or a 1

SDC_CLOSE(#)

Closes the open file where # is 0 or 1

SDC_PUT0 x

Writes a single byte variable x to the open file 0

SDC_PUT1 x

Writes a single byte variable x to the open file 1

x=SDC_GETBYTE(#)

Reads a single byte from the open file number (0 or 1) and stores the value in variable

x, auto increments so the next read will be the next byte in the file.
* is the file number. This must be either a 0 or a 1



Optionally use the SDC_SETPOS() command to set the starting location in the file.

SDC_SETPOS(#,a,b,c,d)

Sets the position in the file to read.

* is the file number (0 or 1)
a,b,c are the Logical sector number (24 bit number of the 256 byte sectors)
a * Most significant byte
b * Mid significant byte
c * Least significant byte
d * The byte in the selected sector (zero based)

For example:

So if you wanted to get the byte 300 in the open file #1 you would use: **SDC_SETPOS(1,0,0,1,43)**

The position would points at the 300th byte. With **n=SDC_GET(1)** so **n** now has the value of the 300th byte, the next **SDCGET(1)** command will get the 301st byte and so on.

A\$=SDC_FILEINFO\$(#)

This will copy the 32 bytes of file info to a string variable such as **A\$** the info can be useful for calculating the file size.

is the file number either 0 or 1.
This is the layout of the bytes in the string:
1-8 File Name
9-11 Extension
12 Attr. bits: \$10=Directory, \$04=SDF Format, \$02=Hidden, \$01=Locked
29-32 File Size in bytes (LSB first)

x=SDC_DELETE(A\$)

Delete a file or empty directory on the SDC

A\$ = variable with the full path to the empty directory or file you want delete.
Result in x where x is:
0 No Error
1 SDC busy too long

```
3 Path name is invalid
4 Miscellaneous hardware error
5 Target file or directory not found
6 Target directory is not empty
```

SDC Commands related to SD card directories

x=SDC_MKDIR(A\$)

Make a directory on the SDC A\$ = variable with the full path to the directory you wish to make Result in x where x is:

```
0 No Error
1 SDC busy too long
3 Path is invalid
4 Miscellaneous hardware error
5 Parent directory not found
6 Name already in use
```

x=SDC_SETDIR(A\$)

Sets the directory on the SDC A\$ = variable with the full path to the directory you change to Result in x where x is:

```
0 No Error
1 SDC busy too long
3 Path is invalid
4 Miscellaneous hardware error
5 Target directory not found
```

A\$=SDC_GETCURDIR()

GET CURRENT DIRECTORY - Retrieves information about the Current Directory for the SD card String variable A\$=Directory info string where the following bytes are:

```
1-8 Filename
9-11 Extension
12-31 Private
```

x=SDC_INITDIR(A\$)

First step to getting a directory listing.

```
To get a directory you must first use this command to setup where and what to list on the directory.
A$ = variable to the full path name of the target directory.
The final component of the path name should be a wildcard pattern that will be used to filter the list
of returned items.
```

Example:

A\$="MYDIR/*.*)"

will list everything in MYDIR

A\$="MYDIR/*.TXT"

will list files ending with .TXT Result in x where x is:

```
0 No Error
1 SDC busy too long
3 Path is invalid
4 Miscellaneous hardware error
5 Target directory not found
```

SDC_DIRPAGE A\$,B\$,x

Second step to getting a directory listing.

This command returns a 256 byte data block which is divided into 16 records of 16 bytes each. Each record describes one item. If there are not enough items to fill the entire page then unused records are filled with zeroes. You may continue to send commands for additional pages until a page containing at least one unused record is returned. Since the directory listing is 256 bytes and the max size of a string is 255 bytes. The listing is split into two string variables with 128 bytes of the directory each.

The first variable **A\$** will get the the first 128 bytes and the second variable **B\$** will get the second 128 bytes of the directory listing. Each entry is:

```
1-8 File Name
9-11 Extension
2 Attribute bits
$10=Directory,
$02 Hidden,
$01 Locked
13-16 Size in bytes (MSB first)
```

Result in x where x is:

```
0 No Error
1 SDC busy too long
4 Listing has not been initiated or already reached the end of a listing
```

SDC Commands for Audio Playback

- **SDC_PLAY** - Playback mono audio samples at 44.75 kHz
- **SDC_PLAY0RC90L** - Playback mono audio samples at 44.75 kHz
- **SDC_PLAY0RC90R** - Playback mono audio samples at 44.75 kHz
- **SDC_PLAY0RC90S** - Playback stereo audio samples at 22.375 kHz
-

SDC_PLAY - Playback an audio file directly stored on the SDC output through the CoCo directly.

Usage:

```
SDC_PLAY"MYAUDIO.RAW"
```



While the sample is playing you can press the BREAK key to stop it.

In order for you to get your audio sample in the correct format to be played back you'll need to prepare your audio samples and put them on the SD card. The format for the raw audio file that will be played is mono 8 bits unsigned. To convert any sound file or even the audio from a video file to the correct format used with the SDCPLAY command use **FFMPEG** and the following command: (Note: all one line)

```
ffmpeg -i source_audio.mp3 -acodec pcm_u8 -f u8 -ac 1 -ar 44750 -af  
aresample=44750:filter_size=256:cutoff=1.0 MYAUDIO.RAW
```



SDC_PLAYORCL & SDC_PLAYORCR use the same audio format as the regular SDC_PLAY command except the output is sent to the Orchestra90/COCOFLASH Left or Right speaker.

If you want to stream 8 bit stereo sound from your CoCo to the COCOFLASH/Orchestra90 use the command:

```
SDC_PLAYORC90S"MYSAMPLE.RAW"
```

MYSAMPLE.RAW is stored on the SD card in your SDC Controller. It is can be created with the **FFMPEG** command below: (Note: all one line)

```
ffmpeg -i source_audio.mp3 -acodec pcm_u8 -f u8 -ac 2 -ar 22375 -af  
aresample=22375:filter_size=256:cutoff=1.0 MYSAMPLE.RAW
```

Floating Point Commands

One of the things that makes a compiler so fast is that it uses integer math. If you must use floating point math and you don't mind the slowdown in speed you can use the following commands.

Command	Description	Command	Description
FLOATADD(FP_X,FP_Y)	Floating Point ADD	FLOATATAN(FP_Y)	Floating Point ATAN
FLOATCOS(FP_Y)	Floating Point COS	FLOATDIV(FP_X,FP_Y)	Floating Point DIV
FLOATEXP(FP_X)	Floating Point EXP	FLOATLOG(FP_X)	Floating Point LOG
FLOATMUL(FP_X,FP_Y)	Floating Point MUL	FLOATSIN(FP_X)	Floating Point SIN
FLOATSQR(FP_Y)	Floating Point SQR	FLOATSUB(FP_X,FP_Y)	Floating Point SUB
FLOATTAN(FP_X)	Floating Point TAN		

Floating Point String conversion commands:

Command	Description	Command	Description
FLOATTOSTR(FP_A)	Floating Point number to a string	STRTOFLOAT(A\$)	Convert a string to a Floating Point variable

Floating Point Comparison commands:

Command	Description	Command	Description
CMPGT(FP_A,FP_B)	Floating Point Compare if Greater Than	CMPGE(FP_A,FP_B)	Floating Point Compare if Greater Than or Equal
CMPEQ(FP_A,FP_B)	Floating Point Compare if Equal	CMPNE(FP_A,FP_B)	Floating Point Compare if Not Equal
CMPLT(FP_A,FP_B)	Floating Point Compare if Less Than or Equal	CMPLT(FP_A,FP_B)	Floating Point Compare if Less Than

In order to use floating point variables you must prefix the variable name with **FP_**, for example:

```
FP_X=FLOATSQR(12.33452)
```

FP_X will now equal 3.51205353

```
FP_Var5=100.12345
```

FP_X and a variable named **X** are *different* variables. **X** will be a signed 16 bit integer and **FP_X** is a floating point number.

Variable conversions can only be done directly as a single command and not in two steps. You cannot do FP functions assigned directly to a signed integer variable:

```
X=FLOATMUL(100,0.100912345)
```

You must do it in two steps, first use a floating point variable with the the math function as

```
FP_Var5=FLOATMUL(100,0.100912345)
```

Results **FP_Var5 = 100.912345**

Then copy the floating point number to the signed integer variable as **X=FP_Var5** then **X** will equal 101 (rounding is done)

You can assign a **FP** number directly to a signed int as: **X=100.912345** then **X** will equal 101 (rounding is done)

X=FP_Var1 then **X** will equal the signed integer value of the floating point variable **FP_Var1**

C(3,6)=FP_Var2 then the array **C(3,6)** will equal the signed integer value of the floating point variable **FP_Var2** Conversion from signed integers to **FP** variables can be done directly as **FP_Var1=X**

If you want to assign an equation of signed ints to a floating point variable it must be done with the **INT()** command **FP_Var1=INT(X*32+Y/5)**

Input values of the commands can be any of the following:

A floating point variable such as **FP_MyFloatVariable1** as

```
FP_Var2=FLOATADD(FP_MyFloatVariable1,FP_Var1)
```

A floating point number such as 100.352 as:

```
FP_Var2=FLOATADD(FP_Var1,100.352)
```

A regular 16 bit signed variable, must use **INT()** as:

```
FP_Var2=FLOATADD(FP_Var1,INT(X))
```

A regular 16 bit signed expression, must use **INT()** as:

```
FP_Var2=FLOATADD(INT(X*23+F),FP_Var1)
```

You can not do complicated equations with floating point math directly. You must do the equation in steps.

Example

if you wanted to do **FP_Var1=FP_Var2*55.234+63.56*X**

You would need to do this as:

```
FP_Temp1=FLOATMUL(FP_Var2,55.234)
FP_Temp2=FLOATMUL(63.56,INT(X))
FP_Var1=FLOATADD(FP_Temp1,FP_Temp2)
```

To convert user input to a floating point number it must be in a string variable and converted to a floating point number with the command **STRTOFLOAT(A\$)** useful for converting user input into float values.

Example:

```
INPUT'ENTER A NUMBER';N$
```

```
FP_Var1=STRTOFLOAT(N$)
```

To do comparisons with Floating point numbers you must use one of:

- **CMPGT(FP_A,FP_B)** - Floating Point Compare if Greater Than
- **CMPGE(FP_A,FP_B)** - Floating Point Compare if Greater Than or Equal
- **CMPEQ(FP_A,FP_B)** - Floating Point Compare if Equal
- **CMPNE(FP_A,FP_B)** - Floating Point Compare if Not Equal
- **CMPLT(FP_A,FP_B)** - Floating Point Compare if Less Than or Equal
- **CMPLT(FP_A,FP_B)** - Floating Point Compare if Less Than

Example:

```
IF CMPGT(FP_Var1,VP_Var6) THEN ?'VP_Var6 is > VP_Var1*
```

These special comparisons must be done on their own after the IF statement. Anything after the first CMPxx(,) will be ignored.

If you wanted to do: **IF CMPGT(FP_Var1,VP_Var6) AND A=B THEN ...**, for example;

You would need to do this as:

```
IF CMPGT(FP_Var1,VP_Var6) THEN IF A=B THEN ...
```

Another example:

```
IF CMPGT(FP_Var1,VP_Var6) OR A=B THEN ...
```

You would need to do this as:

```
IF CMPGT(FP_Var1,VP_Var6) THEN IF A=B THEN ...  
ELSE IF A=B THEN ...
```

Additional Floating Point Details

Printing of floating point numbers directly will display a kind of broken scientific version of the floating point number on screen. You can use the function **FLOATTOSTR(FP_A)** Which cleanly formats a Floating Point number to a string which you can then print on screen. Although the number is still going to display in scientific notation. You can use the code below to show floating point numbers formatted as normal numbers. The variable **V\$** can be manipulated as you want with regular string commands like **MID\$** to format the string as you want for your program.

```
FP_C=FLOATMUL(-234.54321,234.54321)  
FP$=FLOATTOSTR(FP_C)
```

```

' Get the sign of the number
S$=LEFT$(FP$,1)
' Get the numbers without the decimal
N$=MID$(FP$,2,1)+MID$(FP$,4,8)
' Get the Exponent + 1
E=VAL(RIGHT$(FP$,3))+1
SELECT CASE E

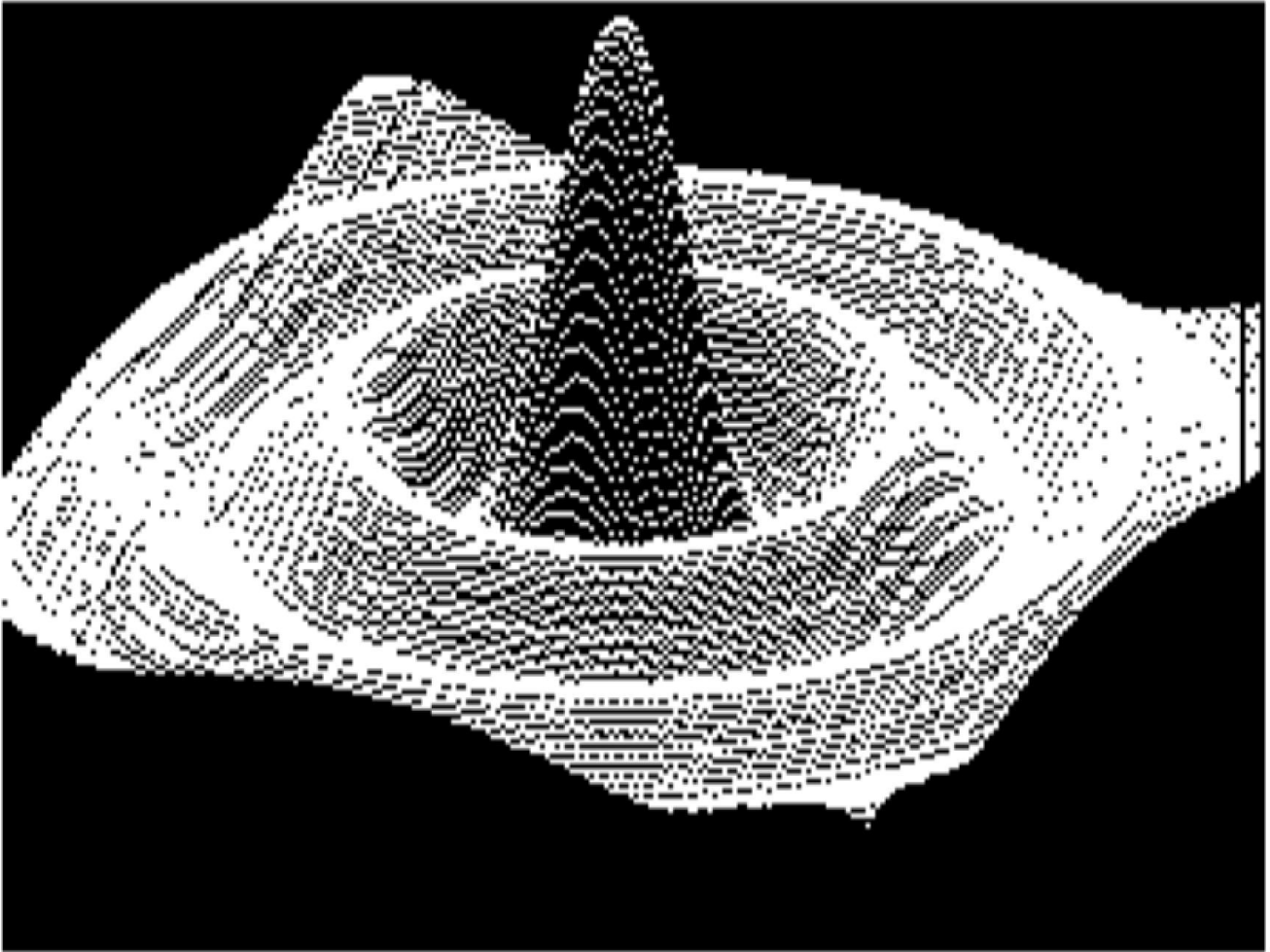
CASE IS <1
V$=S$+"0."+STRING$(-E,"0")+N$
CASE 1 TO 8
V$=S$+LEFT$(N$,E)+"."+RIGHT$(N$,9-E)
CASE IS >8

V$=S$+N$+STRING$(E-9,"0")
End Select
?"FP$=";FP$
?"V$=";V$

```

Output is: **FP\$=-5.50105174E+04 V\$=-55010.5174**

This is a tweaked version of James Diffendaffer's 3D plot program that I converted to working on a CoCo 3 to work on a CoCo 1 & 2



```

0 CX=250:CY=192:PMODE 4,1:PCLS:SCREEN 1,1
1 DIM R(250):FOR I=0 TO CX:R(I)=CY:NEXT I:GOTO 10
2 R=SQR(X*X+Y*Y)*1.5: IF R=0 THEN F=90:GOTO 4
3 F=90*SIN(R)/R
4 A=10*X+125-5*Y:B=5*Y+2.5*X+93:RETURN
10 FOR Y=10 TO -10 STEP -0.1
70 FOR X=10 TO -10 STEP -0.1
80 GOSUB 2
82 IF A<0 THEN A=0
83 IF A>255 then A=255
84 IF R(A)>B-F THEN R(A)=B-F:*PSET*(A,B-F)
90 NEXT X,Y
101 GOTO 101

```

Below is a version of the same program but ready to be compiled with the new floating point commands. Note that you can't use floating point numbers with the FOR NEXT commands so this is a work around.

```

0 CX=250:CY=192:PMODE 4,1:PCLS:SCREEN 1,1
1 DIM R(250):FOR I=0 TO CX:R(I)=CY:NEXT I:GOTO 10
2 'R=SQR(X*X+Y*Y)*1.5: IF R=0 THEN F=90:GOTO 4

```

```

FP_Temp1=FLOATMUL(FP_X,FP_X)
FP_Temp2=FLOATMUL(FP_Y,FP_Y)
FP_Temp1=FLOATADD(FP_Temp1,FP_Temp2)
FP_R=FLOATSQR(FP_Temp1)
FP_R=FLOATMUL(FP_R,1.5)
IF CMPEQ(FP_R,0) THEN FP_F=90:GOTO 4

3 'F=90*SIN(R)/R

FP_Temp1=FLOATSIN(FP_R)
FP_Temp1=FLOATMUL(90,FP_Temp1)
FP_F=FLOATDIV(FP_Temp1,FP_R)
F=FP_F

4 'A=10*X+125-5*Y:B=5*Y+2.5*X+93:RETURN

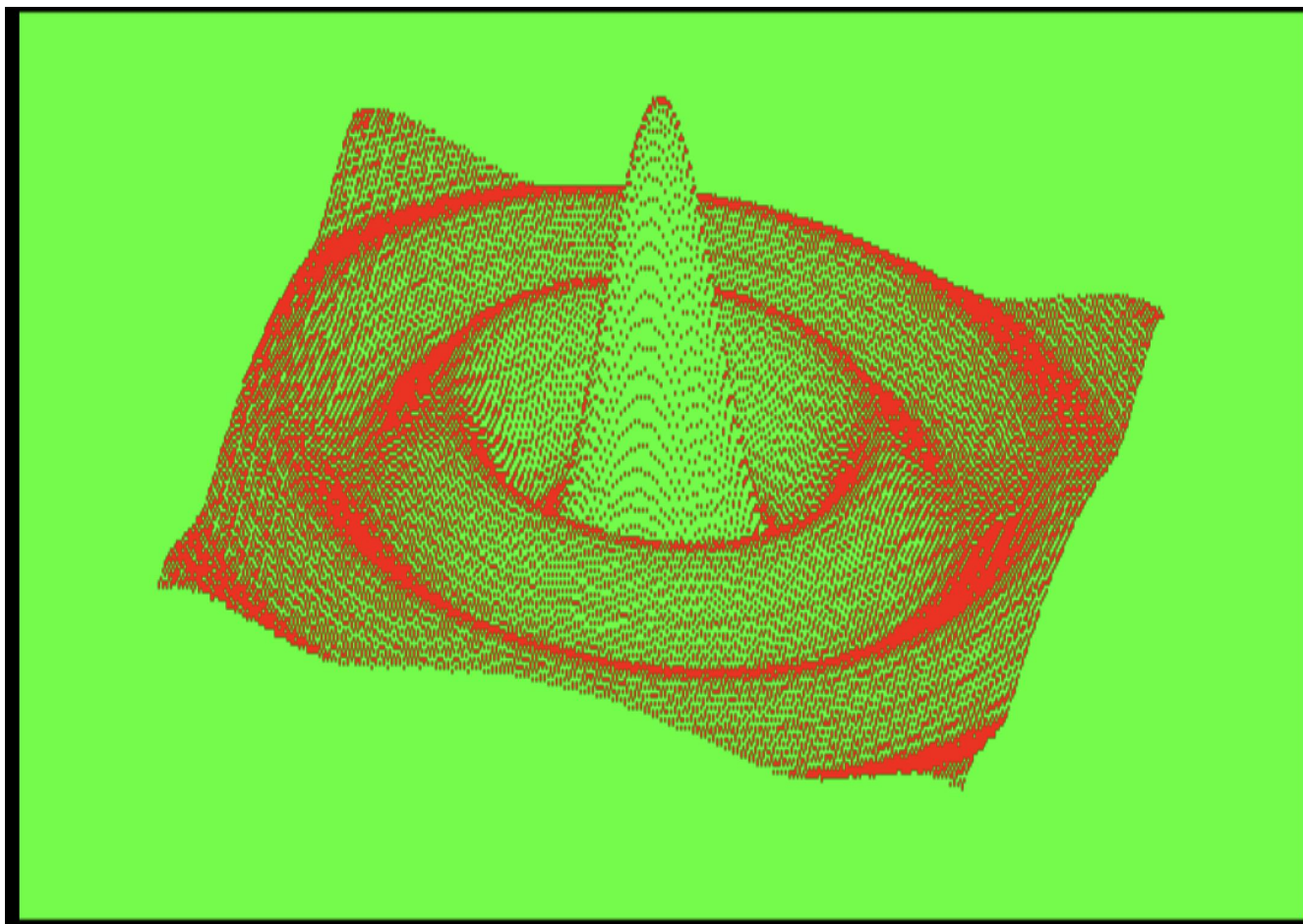
FP_Temp1=FLOATMUL(10,FP_X)
FP_Temp2=FLOATMUL(5,FP_Y)
FP_A=FLOATADD(FP_Temp1,125)
FP_A=FLOATSUB(FP_A,FP_Temp2)
A=FP_A
FP_Temp1=FLOATMUL(5,FP_Y)
FP_Temp2=FLOATMUL(2.5,FP_X)
FP_B=FLOATADD(FP_Temp1,FP_Temp2)
FP_B=FLOATADD(FP_B,93)

B=FP_B

RETURN
10 FOR Y=100 TO -100 STEP -1
70 FOR X=100 TO -100 STEP -1
FP_Y=FLOATDIV(INT(Y),10)
FP_X=FLOATDIV(INT(X),10)
80 GOSUB 2
82 IF A<0 THEN A=0
83 IF A>255 then A=255
84 IF R(A)>B-F THEN R(A)=B-F:*PSET*(A,B-F)
90 NEXT X,Y
101 GOTO 101

```

This is the same program scaled for the highest resolution screen a CoCo 3 can use.



```

0 CX=640:CY=224:GMODE 156,1:GCLS:SCREEN 1,1
1 DIM R(640):FOR I=0 TO CX:R(I)=CY:NEXT I:GOTO 10
2 'R=SQR(X*X+Y*Y)*1.5: IF R=0 THEN F=90:GOTO 4
FP_Temp1=FLOATMUL(FP_X,FP_X)
FP_Temp2=FLOATMUL(FP_Y,FP_Y)
FP_Temp1=FLOATADD(FP_Temp1,FP_Temp2)
FP_R=FLOATSQR(FP_Temp1)
FP_R=FLOATMUL(FP_R,1.5)
IF CMPEQ(FP_R,0) THEN FP_F=90:GOTO 4
3 'F=90*SIN(R)/R
FP_Temp1=FLOATSIN(FP_R)
FP_Temp1=FLOATMUL(90,FP_Temp1)
FP_F=FLOATDIV(FP_Temp1,FP_R)
F=FP_F
4 'A=10*X+125-5*Y:B=5*Y+2.5*X+93:RETURN
'A=20*X+319-5*Y:B=5.895*Y+2.5*X+111:RETURN
FP_Temp1=FLOATMUL(20,FP_X)
FP_Temp2=FLOATMUL(5,FP_Y)
FP_A=FLOATADD(FP_Temp1,319)
FP_A=FLOATSUB(FP_A,FP_Temp2)
A=FP_A
FP_Temp1=FLOATMUL(5.895,FP_Y)
FP_Temp2=FLOATMUL(2.5,FP_X)
FP_B=FLOATADD(FP_Temp1,FP_Temp2)

```

```
FP_B=FLOATADD(FP_B,111)
B=FP_B
RETURN
10 FOR Y=100 TO -100 STEP -1
70 FOR X=100 TO -100 STEP -1
FP_Y=FLOATDIV(INT(Y),10)
FP_X=FLOATDIV(INT(X),10)
80 GOSUB 2
82 IF A<0 THEN A=0
83 IF A>639 then A=639
84 IF R(A)>B-F THEN R(A)=B-F:SET(A,B-F,1)
90 NEXT X,Y
101 GOTO 101
```

Get Command

- Dimension Size Calculation

To calculate the size of the array space for your GET/PUT buffer use the following formula:

First dimension in the array is calculated with this formula:

```
(INT(Width in pixels/8)+3) * 8
```

Second dimension in the array is simply the number of rows in your sprite

For example, if you have a sprite that is 15 pixels wide and 9 rows high such as:

```
GET(0,0)-(14,8),Sprite1,G
```

The calculation for the needed space is:

```
(INT(15/8)+3)*8 = 32
```

The DIM command for this array would be:

```
DIM Sprite1(32,9)
```

If the calculated value for the first dimension of your GET buffer array is larger than 254 you will need to use these values for your array * (INT(Width in pixels/8)+3) * 4 , Height in Pixels * 2 *

```
GET(0,0)-(255,3),Sprite1,G
```

The calculation for the needed space is: * (INT(256/8)+3)*4 = 140 , 4 * 2 = 8 *

The DIM command for this array would be:

```
DIM Sprite1(140,8)
```

The reason so much space is needed for the GET buffer is because the GET command preprocesses the sprite data and saves bit shifted versions in the array space that are ready to be PUT on the screen as fast as possible. This means sprites will be just as fast on a byte boundary as it is on any other pixel.

Graphics commands:

GMODE ModeNumber,GraphicsPage

Selects the graphics screen and the graphics page.

ModeNumber is the graphics mode you want to use

GraphicsPage is the Page you want to show/use for your graphics commands



To see a list of ModeNumbers and the resolutions go here [Special](#) note the ModeNumber must be an actual number and cannot be a variable as the compiler needs to know exactly which graphic mode commands to be included at compile time. GraphicsPage can be a variable. If you are going to use Graphic pages, the compiler needs to know how many pages to reserve in RAM (for CoCo 1 & 2 graphics). So you must have a GMODE #,MaxPages entry at the beginning of your BASIC program. Where the value of MaxPages will be an actual number and not a variable.

GCLS #

Colour the graphics screen

is the colour value you want the screen to be coloured

GCOPY SourcePage,DestinationPage

Makes a copy the Source graphics page to the Destination graphics page.

SourcePage - Source graphics page

DestinationPage - Destination graphics page

NTSC_FONTCOLOURS(B,F)

Background, Foreground - Sets the CoCo 3 NTSC font colours

Background - Background colour number

Foreground - Foreground colour number

SET(x,y,Colour)

Sets a pixel on the screen

x,y - Screen location of the pixel to be drawn

Colour - Colour Number of the pixel to be drawn

POINT(x,y)

Returns the colour value of the pixel selected

x,y - Screen location of the pixel value requested

LINE(x0,y0)-(x1,y1),Colour[,B][F]

Draw a line

```
x0,y0 - Starting location
x1,y1 - Ending location
Colour -Colour of the Line or Box to draw
  B - Draw a Box
  F - Fill the Box
```

PAINT(x,y),OldColour,FillColour

Fills the old colour value with the fill colour value which must also be the border colour of the section you are painting

```
x,y - Starting location
OldColour - Colour Number
FillColour - Colour Number
```

CIRCLE(x,y),Radius,Colour

Draws a circle on the screen x,y

```
x,y - Origin of the circle
Radius - Size of the circle, to keep the aspect ratio close to round Some of the graphics modes use
scaling so the Radius isn't always a count of actual pixel values.
Colour - Colour Number of the circle to draw
```



CIRCLE is currently unable to draw ARCS.

PALETTE v,Colour

Sets the CoCo 3 Palette value

```
v - palette slot of 0 to 15
Colour - Colour value of 0 to 63
```

DRAW "line string"

Draws a line string on the screen. The line string is a string of characters that represent the line to be drawn. The general characters are:

letter	instruction
M	start moving position
B	move but draw blank
u	draw upward (north)
E	draw up and right (northeast)
R	draw rightward (east)

F	draw down and right (southeast)
D	draw downward (south)
G	draw down and left (southwest)
L	draw leftward (west)
H	draw up and left (northwest)
N	draw no position update
C	color choice
A	angle
s	scale
X	execute substring and return

Example: **DRAW "C2BM80,90R100U50L50D50"**



GET & PUT commands are not yet available

Sprite Handling - Overview

To use sprites in your BASIC programs you need to first prepare them with some external tools.

- A graphic editor like GIMP to make or convert/export your sprite image into a 32 bit RGBA (includes transparency) PNG file.
- The command line tool (included with this compiler) PNGtoCCSB to convert the PNG file into a Sprite to be used in your BASIC program.

Typically you'll use PNGtoCCSB to convert your PNG file into a sprite with a command similar to:

```
PNGtoCCSB -g15 -s0 MySprite.PNG ` `PNGtoCCSB -g15 -s0 MyOtherSprite.PNG
```

This tool will convert the PNG files **MySprite.PNG** & **MyOtherSprite.PNG** into compiled sprites ready to be used.

The **-g15** option tells the tool we are going to use this sprite with a **GMODE 15** graphics screen in your BASIC program. When converting the PNG to a compiled sprite it will match the colours of the PNG to the available colours in the GMODE & Screen mode (-sx option) on the CoCo.

This will save the compiled sprites as **MySprite.asm** & **MyOtherSprite.asm**

In your BASIC program you will need to start your program with the following commands:

```
Sprite_Load "MySprite.asm", 0 ' Load sprite as #0
Sprite_Load "MyOtherSprite.asm", 1 ' Load sprite as #1
GMODE 15,1 'This will reserve graphics page 0 and graphics page 1
GMODE 15,0 'Set the current graphics page to use as 0
GCLS 0 'Colour the graphics screen colour 0
SCREEN 1,0 'Show the graphics screen (you could do this later)
```

Draw any background graphics for your program here...

Once your background graphics are drawn you need to copy screen 0 to screen 1 this is done with the GCOPY command. This is necessary as the sprite routines use double buffering where the screen is switched to show screen 1 the sprite updates are drawn on screen 0, once the sprites are updated, screen 0 is shown and screen 1 sprite updates are drawn.

```
`GCOPY 0,1 'Copy graphics screen 0 to graphics screen 1`
```

Now that your background is setup you are ready to draw the sprites on the screen using the steps outlined below.



A normal Sprite usage routine would look like this:

```
SPRITE LOCATE 0, X1, Y1 ' Set the location of sprite 0
SPRITE BACKUP 0 ' Copy what's behind sprite 0
SPRITE SHOW 0, 0 ' Draw sprite 0, frame 0
SPRITE LOCATE 1, X2, Y2 ' Set the location of sprite 1
SPRITE BACKUP 1 ' Copy what's behind sprite 1
SPRITE SHOW 1, 0 ' Draw sprite 1, frame 0
```

Except for the **SPRITE LOCATE** command the above commands aren't executed at this point, they are added to a sprite command queue that is executed when the **WAIT VBL** command is used as:

`WAIT VBL` This is when the actual sprite updates occur`

Typically you will next want the sprite erase commands after the Wait VBL command, which again are only added to the sprite command queue at this point. It is usually best to reverse the order of the sprites for erasing.

```
SPRITE ERASE 1  Erase Sprite 1 (reverse order is good)
SPRITE ERASE 0  Erase Sprite 0
```

CoCo 1 & 2 Graphic Modes

GMODE #	Resolution	Colours	Bytes/Screen	Mode Name
0	32 x 16	9	512	internal alphanumeric
1	32 x 16	2	512	external alphanumeric
2	32 x 16	9	512	semi-graphic-4
3	64 x 32	9	2048	semi-graphic-8
4	64 x 48	9	512	semi-graphic-6
5	64 x 48	9	3072	semi-graphic-12
6	64 x 64	9	2048	semi-graphic-8
7	64 x 96	9	6144	semi-graphic-12
8	64 x 192	9	1024	semi-graphic-24
9	64 x 64	4	1024	full graphic 1-C
10	128 x 64	2	2048	full graphic 1-R
11	128 x 64	4	1536	full graphic 2-C
12	128 x 96	2	3072	full graphic 2-R
13	128 x 96	4	6144	full graphic 3-C
14	128 x 192	2	6144	full graphic 3-R
15	128 x 192	4	6144	full graphic 6-C
16	256 x 192	2	6144	full graphic 6-R
17			6144	Direct Memory Access

COCO 3 Graphic Modes

GMODE #	Resolution	Colours	Bytes/Screen
100	64 x 192	4	3200
101	64 x 200	4	3200

GMode #	Resolution	Colours	Bytes/Screen
102	64 x 225	4	3600
103	64 x 192	16	6144
104	64 x 200	16	6400
105	64 x 225	16	7200
106	80 x 192	4	3840
107	80 x 200	4	4000
108	80 x 225	4	4500
109	80 x 192	16	7680
110	80 x 200	16	8000
111	80 x 225	16	9000
112	128 x 192	2	3072
113	128 x 200	2	3200
114	128 x 225	2	3600
115	128 x 192	4	6144
116	128 x 200	4	6400
117	128 x 225	4	7200
118	128 x 192	16	12288
119	128 x 200	16	12800
120	128 x 225	16	14400
121	160 x 192 *	2	3840
* (viewable) really 128x192, It's a special mode that repeats the left 4 bytes on the right side of the screen			
122	160 X 200 *	2	4000
* (viewable) really 128x192, It's a special mode that repeats the left 4 bytes on the right side of the screen			
123	160 x 225 *	2	4500
* (viewable) really 128x192, It's a samplepecial mode that repeats the left 4 bytes on the right side of the screen			
124	160 x 192	4	7680
125	160 x 200	4	8000
126	160 x 225	4	9000
127	160 x 192	16	15360
128	160 x 200	16	16000
129	160 x 225	16	18000
130	256 x 192	2	6144
131	256 x 200	2	6400
132	256 x 225	2	7200
133	256 x 192	4	12288

GMODE #	Resolution	Colours	Bytes/Screen
134	256 x 200	4	12800
135	256 x 225	4	14400
136	256 x 192	16	24576
137	256 x 200	16	25600
138	256 x 225	16	28800
139	320 x 192	2	7680
140	320 x 200	2	8000
141	320 x 225	2	9000
142	320 x 192	4	15360
143	320 x 200	4	16000
144	320 x 225	4	18000
145	320 x 192	16	30720
146	320 x 200	16	32000
147	320 x 225	16	36000
148	512 x 192	2	12288
149	512 x 200	2	12800
150	512 x 225	2	14400
151	512 x 192	4	24576
152	512 x 200	4	25600
153	512 x 225	4	28800
154	640 x 192	2	15360
155	640 x 200	2	16000
156	640 x 225	2	18000
157	640 x 192	4	30720
158	640 x 200	4	32000
159	640 x 225	4	36000
160	512 x 192	256	24576
161	512 x 200	256	25600
162	512 x 225	256	28800
163	640 x 192	256	30720
164	640 x 200	256	32000
165	640 x 225	256	36000



160 - 165 are the new NTSC composite video out modes for the CoCo3, these are new GMODE values from 160 to 165

Limitations of the Compiler

- Other than support for **LOADM** it can't handle Disk access, but you can access the **CoCoSDC** for many disk type functions.
- **CIRCLE** command can only draw complete circles, you can't squeeze them or draw an arc

Error Handling

If your program isn't compiling, a lot of the times it's because the compiler is having a hard time parsing the program. Usually, making sure you have spaces between commands and variables and operators and variables.

You can often identify the source of an issue by examining the error message and the line number provided, which indicate where the compiler encountered a problem. If your program does not include line numbers, the error message will not be able to specify the exact line where the error occurred.

Also looking at the end of the actual `.asm` file it created might help to see what the compiler is trying to parse and failed.

Thanks

- I'd like to thank Scott Cooper (Tazman) for initial testing of the compiler.
- I'd also like to thank others on Discord who inspired me to keep adding new features, including Bruce D. Moore, Erico Monteiro.
- Typesetting of PDF documentation, Pete Willard

Change Log

For more info check out the blog post here: <https://wordpress.com/post/nowhereman999.wordpress.com/5054>

V 4.41

- Added the DRAW command for all graphics modes including the semi-graphics modes
- Improved the handling of single line IF commands that have THEN or ELSE with a line number and not a GOTO line number
- Fixed POKE command so it can now handle math functions directly for both the address and the value to poke
- Fixed a bug with LINE command with horizontal lines with two colours, depending on the offset it could draw the line smaller than it should be
- Updated SDECB.bas IDE to ignore TIMER, it can now be treated as a variable to be set just as on Extended Color BASIC

V 4.40

- Added -dragon option, which will output a compiled program that will work for the Dragon computers keyboard layout instead of the CoCo keyboard layout
- Now if coco1/2 or dragon mode, and no drive the NMI IRQ is set to RTI

V 4.39

- Fixed SCREEN and GMODE commands so that the graphic screen is only shown with the SCREEN 1 command now and pointers for screen locations are set properly
- Fixed an issue with IF/THEN/ELSE in certain conditions

V 4.38

- Tweaked the COPYBLOCKS command, if there is a 6309 in the CoCo it will use TFM to do the copy (way faster)
- Fixed a bug where the SCREEN command wasn't going back to text mode or graphics mode when sprites are being used

V 4.37

- Added TRIM\$,LTRIM\$ & RTRIM\$ commands that will remove the spaces in a string
- Fixed a bug where the IRQ wasn't being set properly if you used PLAY and a CoCo 3 GMODE

V 4.36

- Added command CPUSPEED # where # is 1 for .895 Mhz, 2 for 1.79 Mhz, 3 for 2.8636 MHz and anything else will set the speed to max for that computer's hardware. Which also will put a 6309 in native mode.
- Added command COPYBLOCKS, which copies 8k blocks using stack blasting useful for CoCo3 copying from

buffer 0 to buffer 1

- Fix a bug with deep IF/THEN/ELSE layers that could exit to the wrong location
- Tweaked the DIV16 & DIV16Rounding code so that the resulting value in D will be reflected in the condition codes when exiting the routine

v4.35 - Fixed a problem with handling DATA statements that had many commas in a row like (DATA "THIS" ,,,2,,)

V4.34 - Can now handle 10000 IF/THEN/ELSE commands per program, was previously set at 100 - Fixed a bug detecting string array variables - Made STRING\$ function now handle ascii codes instead of only string values for the 2nd value in the command - Fixed a bug with MID\$ - can now handle 1000 deep expression comparisons, was limited to 10 previously - Fixed a bug with IF/THEN/ELSE where compiler could re-use the same labels - Fixed a bug with numeric arrays with one element (wasnt pointing at the correct RAM location) - Fixed a bug with the SDC_GETCURDIR\$ command - Fixed a bug with the PLAY command (IRQ was returning to the wrong location) - PLAY command now automatically plays at normal speed then automatically speeds up the CoCo after playing

V4.33 - If your CoCo 3 can support 2.8 Mhz high speed the compiled code now runs at triple speed (2.8 Mhz) for most operations (not disk, etc.)

V4.32 - Added new command SDC_BIGLOADM, for fast loading the CoCo 3 memory banks, useful for loading game backgrounds or Other large amounts for data - Added support for Drive number 0 or 1 for the SDC_Play commands

V4.31 - Fixed a bug in the SDC_LOADM command that wasnt loading from drive # 0 properly

V4.30 - Fixed a couple bugs with the LOADM command (wasnt finding the last granule properly if the file used granules beyond \$40)

V4.29

- Added support for the NTSC composite video out modes (256 colours) for the CoCo3, these are new GMODE values from 160 to 165 - See the updated manual for more info
- Added new command NTSC_FONTCOLOURS b,f to set the background and foreground colours of the fonts used with the new NTSC composite output GMODEs

V4.25

- Fixed a bug with with using numeric commands like RNDZ(x) before an close bracket of a graphics command like SET(x,y,RNDZ(3))

V4.24

- Fixed a bug in the compiler when it was assigning the value of an equation to a variable. If the equation didn't have a variable and it was doing AND, OR, XOR or MOD it was not setting the variable properly. It also wasn't handling HEX values properly in this same routine.

V4.23

- Made the Tokenizer a little more robust, it now detects array names better if there aren't spaces before the array name and the array is being used in an equation.

V4.22

- Fixed a bug with large BASIC programs especially if it used a large amount of space for the ADDASSEM: section

V 4.21

- Fixed a bug where the PROGRAM start was not being setup to the correct address when GMODE was being used. Thanks to Tazman (Scott Cooper) for finding the bug.

V 4.20

- Fixed a bug in GMODE 1 graphic commands (This mode only supports two colours, not 9 like GMODE 0)
- Added printing to the screen using the semi-graphic modes, the semi-graphics modes use the built in VDG font except for GMODE 4 which uses SG6 and the built in fonts aren't supported in this mode. So the font that is shown is a large 6x6 matrix font. Now every GMODE can now print text on the screen using LOCATE x,y:PRINT #-3,"Hello World"

V 4.11

- Can now handle MID\$(String,Start) which will copy the String starting at location Start copying the rest of the string. It no longer needs to be MID\$(String,Start,Length).

V 4.10

- Added text fonts that can be used directly in any graphics mode including CoCo 3 modes but not the semi graphic modes Use **LOCATE x,y** and **PRINT #-3,"Hello World"** to print to the graphics screen. You must select the font using the **-f** option. The font names end with **_Bx_Fx** where Bx is the Background colour and Fx is the Foreground colour. **Example: -fCoCoT1_B0_F2**
- Can now put compiler options on the first line of the program using **' CompilerOptions: -**
Example: 'CompileOptions: -fArcade_B0_F1 -o -s32

V 4.03

- Fixed a bug in the **GCOPY** command
- Tweaked the code for the **GMODE** command

V 4.02

- Numeric array sizes can now be larger than 255, it will now accept **DIM A(1000)** or **DIM MyArray(300,2,2)**
- String array sizes can now be larger than 255, it will now accept **DIM A\$(1000)** or **DIM MyArray\$(300,2,2)**, the default size for strings (which includes array elements) is 255 therefore you must use the compiler option **-sxxx** where xxx is the size of the string and make it a smaller string size if you are going to use many string arrays

V 4.01

- With the **GMODE ModeNumber,GraphicsPage** the compiler can now handle variables for the *GraphicsPage* value. The *ModeNumber* value will always need to be an actual number
- **PALETTE** command now will wait for a vsync to change the value (should make sure there's no sparkels)

V 4.00

- Added New command **GMODE** which allows you to easily select every graphic mode the CoCo 1,2 or 3 can produce
- Added New command **GCOPY** which allows you to copy graphic pages to other graphic pages
- Tweaked **LINE**, **CIRCLE** & **PAINT** commands to use all the available graphic modes and colours of the CoCo 1,2 or 3

V 3.03

- Tweaked the **AND**, **OR**, **XOR** commands as the optimizer was not handling them properly

V 3.02

- Fixed a bug with optimiing when it is doing exponents

V 3.01

- Fixed a small bug with Printing of double quotes ""

V 3.00

- Added SDC file support, you can now read and write files on the SDC filesystem directly
- If a 6309 is present it puts it in native mode (should speed up code about 15%)
- Added compiler option **-a** which makes your program autostart
- Fixed a bug where if a variable name had **AND, OR, MOD, XOR, NOT** or **DIVR** in the name it would create variable names and operators out of it
- Fixed a bug with IF command with multiple numeric commands that weren't evaluating properly

V 2.22

- Fixed a bug with a comment at the end of a **DATA** statement, it was sometimes not being ignored

V 2.21

- Fixed a bug handling close brackets

V 2.20

- Fixed handling of direct number conversion of a number that starts with a minus sign
- Fixed handling of **DATA** values that start with a minus sign
- Matched how BASIC converts a float to an INT (negative numbers always get the integer -1)
- Fixed a bug in multidimensional arrays

V 2.19

- Fixed **LINE-(x,y),PSET** command

V 2.18

- Fixed assigning a variable to numeric commands

V 2.17

- Added **CMPNE(FP_A,FP_B) * Floating Point Compare if Not Equal**

V 2.16

- Found a problem with comparing floating point numbers where the exponent was not being compared correctly
- Tweaked getting the 2nd value in a floating point number if it was a typed value
- Tweaked getting numeric value in as a floating point number can now handle the value if it starts with a decimal
- Tweaked the identification of a label and a variable

V 2.15

- Added limited Floating Point support by integrating the Floating point library * by Lennart Benschop
- Fixed a bug compiling single lines with **IF THEN ELSE**

V 2.14

- Added more commands for streaming audio directly off the CoCo SDC: **SDCPLAYORCS**, **SDCPLAYORCL** & **SDCPLAYORCR** These commands require either an Orchestra 90 or CocoFLASH (<https://www.go4retro.com/products/cocoflash/>) cartridge The **SDCPLAYORCS** command streams stereo 8 bit unsigned audio at 22,375hz to both the left and right channels The **SDCPLAYORCL** streams mono 8 bit unsigned audio at 44,750hz to the Left channel output The **SDCPLAYORCR** streams mono 8 bit unsigned audio at 44,750hz to the Right channel output
- Added version checking if the user is going to have SDC streaming commands like **SDCPLAY** as the

SDC needs version 127 or higher

- Fixed handling of **REM's that were commenting out actual code, it now display better in the `.asm` file** It was also causing errors as some of the command values were printing as ASCII LF/BREAK/Carriage return symbols

V 2.13

- Added **SDCPLAY** command to let you play audio files directly off the CoCo SDC

V 2.12

- Added **GET** and **PUT** commands, added option for *PUT* command to use the usual **PSET** (default), **PRESET**, **AND**, **OR**, **NOT** but also added **XOR**

V 2.11

- Tweaked **PLAY** command, had problems with Quoted PLAY string

V 2.10

- Added **PLAY** command and tweaked **DRAW** command so it handles **;X** with a string if the DRAW command doesn't end with a semi colon
- Fixed a NASTY bug where certain bytes in **ADDASSEM/ENDASSEM** would write over actual program code

V 2.09

- Fixed a bug with **INKEY\$**

V 2.08

- Broke some regular printing with the printing to graphics screen, this is now been fixed
- Fixed a bug with getting the value of an expression before an open bracket if an array was before the open bracket

V 2.07

- Fixed a bug with **REM** or **'** at the end of a line that has variables in it, the line was past the variable name was being ignored by the parser

V 2.06

- Added feature to print to the PMODE 4 screen using **PRINT #-3,"Hello World"**, use **LOCATE x,y** to set the location on screen of the text
- Select which font from the commandline using option **-fxxxx** where xxxx is either **Arcade** or **CoCoT1**

V 2.05

- Added printing to the serial port with **PRINT #-2,**

V 2.04

- Added Disk I/O access commands in the file `Disk_Command.asm`
- The Compiler can now do a **LOADM** command

V 2.03

- Added **-v** (version) and **-h** (help) commandline options, now uses **-V** for verbose level

V 2.02

- Fixed issue with **INPUT** command with quotes and numeric commands being identified as numeric arrays

V 2.01

- Added option **-k** to allow the user to keep the temporary files that are generated when the compiler

is processing the **.BAS** program. The default is to always delete these temp files.

V 2.00

- Can now handle many more variables
- Tokenizer can now parse code better, but still will require spaces in some cases in order to distinguish between commands and variables
- Internally it's now a lot easier to add new BASIC commands.
- Drawing horizontal lines has been sped up a lot, this also speeds up drawing Boxes and Filling boxes with the ,B or ,BF options for the **LINE** command
- Handling of **DATA** commands is improved, it can handle multiple DATA : DATA commands on a single line and it can now handle string values without quotes, just as the CoCo will
- Compiler now has 3 programs to compile before you can use it. Load them into QB64 and compile them in the same folder as the other folders in the .zip file (Basic_Includes & Basic_Commands)
- The three programs are: **BasTo6809.bas**, **BasTo6809.1.Tokenizer.bas** & **BasTo6809.2.Compile.bas**
- Once all three are compiled start the compiler with the usual options as (or similar to):
./BasTo6809 -ascii -v0 -o2 -b0 YourProgram.bas

V 1.19

- The program now closes files #1 and #2 before killing the old one and renaming the Temp.txt to the correct .asm filename when removing unnecessary lines, this was causing file permission errors on Windows OS machines

V 1.18

- Fixed **ADDASSEM** and **ENDASSEM** which are now working correctly
- Added option and changed the code to allow the user to select the Max size of string arrays, max will be 255 but the user should be able to set it lower to save space

V 1.17

- Fixed numeric and String arrays with multi dimensions, wasn't storing the values properly
- Renamed include file **PRINT.ASM** to **Print.asm**
- Fixed VAL command, would return with zero if the string started with a space before the number
- Fixed the Print command, was ignoring , or ; after a string variable