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**, 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 lower-level language for faster execution or to speed up development of assembly language code.

Version Information

BasTo6809 Version: 4.21

Author: Glen Hewlett

GitHub: BASIC-To-6809

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 lot).

-a
: Makes the program autostart after it is loaded

-**v**
: 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 $0E00.

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 uses 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 graphics screen (default is Arcade_B0_F1). Look in folder Basic_Includes/GraphicsMode/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.

Cool things the Compiler can do

– 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, 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 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 replace 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
Colour - Colour value of 0 to 63

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**
– **SDC_PLAY** Command that plays an audio sample or song directly off the SD card in the SDC Controller. See here for more info
– **SDC_PLAYORC90L**, **SDC_PLAYORC90R**, **SDC_PLAYORC90S** these commands are similar to SDCPLAY except the audio is sent to the Orchestra 90 or COCOFLASH cartridge. See here for more info
– **SDC** file access commands that allow you to Read & Write files directly on the SD card's own filesystem. See here for more info
– **Floating Point commands** (special commands to handle floating point calculations and operations. See here for more info
– **GETJOYD** - Quickly get the joystick values of 0,31,63 of both joysticks both horizontally and vertically
- **PRINT #-3,** - Print to the graphics screen created with the GMODE command. Can also be used as ?#-3,"Hello World!". The default font is ArcadeArcade_B0_F1, but you can select others using the compiler command -fxxxx Where xxxx is the font name used for printing to the graphics screen (default is Arcade_B0_F1). Look in folder Basic_Includes/GraphicsMode/Graphic_Screen_Fonts to see font names available

How to use

The compiler is called BasTo6809 and is written in BASIC specifically QB64pe (Phenix Edition). QB64pe is multi-platform so BasTo6809 can be used on a Mac, Linux or Windows machine. You'll need to compile it using QB64pe.

Once you have BasTo6809 compiled using QB64pe you'll need to have it along with a specific folder called Basic_Includes of .asm libraries that the compiler will add (if needed) to the output.asm file it generates from the source .BAS file.

The 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.

Once you're compiling folder is setup 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 -9bl -p cd -o./ML.bin BASIC.asm > ./Assembly_Listing.txt
```

Using **Windows**:

```
.\BasTo6809 -ascii BASIC.bas
lwasm -9bl -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.

Get your system ready to compile BASIC programs for the CoCo

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 and you've used QB64 to compile BasTo6809.bas,BasTo6809.1.Tokenizer.bas and BasTo6809.2.Compile.bas along with cc1sl.bas, you're ready to go.

Make sure your working directory now has the BasTo6809 executable, BasTo6809.1.Tokenizer executable, BasTo6809.2.Compile executable and the cc1sl executable program and the sub folder called "Basic_Includes" which has .asm files and more in it.

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 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.

Next step is to use LWASM to assemble the program into a CoCo executable program, something like this:

```
lwasm -9bl -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 the compiler option -b1. LWASM will take awhile 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 -9bl -p cd -o./HELLO.BIN HELLO.asm > ./NEW_Assembly_Listing.txt
./cc1sl -l HELLO.BIN -oBIGFILE.BIN
```

For Windows:

```
.\BasTo6809 -b1 HELLO.BAS
lwasm -9bl -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

64k programs

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 cc1sl.bas that you must compile with QB64pe before using.

Supported BASIC commands

AUDIO ON/OFF
BUTTON
CIRCLE
CLEAR - Only clears all the variables to zero
CLS
CASE
DATA
DEF FN
DIM
DO (WHILE/UNTIL)
ELSE
ELSEIF
END
END IF
END SELECT
EVERYCASE
EXEC
EXIT (DO,FOR,WHILE)
FOR/NEXT
GCLS - Colour the screen
GETJOYD - New command which allows you to quickly 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:

```
LEFT LEFT RIGHT RIGHT
VERT HORIZ VERT HORIZ
$15A $15B $15C $15D
```

GMODE - Set the graphics mode and graphics page
GOSUB
GOTO
GCOPY - Copy graphics page to another graphics page
IF
INPUT
LET
LINE
LOADM
LOCATE - Set the position on the graphics screen to print text from

LOOP (WHILE/UNTIL)
MOTOR ON/OFF
NEXT
ON GOSUB
ON GOTO
PAINT
PALETTE - Palette v,colour where v is the palette slot of 0 to 15 and colour is the colour value of 0 to 63
PLAY
POINT
POKE
PRINT – Can’t do PRINT USING
READ
RESET
RESTORE
RETURN
SCREEN
SDC_CLOSE(#)
SDC_DELETE(A$)
SDC_DIRPAGE A$,B$,x
SDC_FILEINFO$(#)
SDC_GETBYTE(#)
SDC_GETCURDIR()
SDC_INITDIR(A$)
SDC_LOADM"FILENAME.BIN",#[,Offset]
SDC_MKDIR(A$)
SDC_OPEN”FILENAME.EXT”,”X”,#
SDC_PLAY,SDC_PLAYORC90L,SDC_PLAYORC90R,SDC_PLAYORC90S
SDC_PUT0 x
SDC_PUT1 x
SDC_SAVEM"FILENAME.BIN",#,Start,End,Exec
SDC_SETDIR(A$)
SDC_SETPOS(#,a,b,c,d)
SELECT
SET
STEP
STOP
SOUND
TAB()
TIMER

UNTIL
WHILE/WEND
WPOKE

Numeric Commands it can handle

ABS()
ASC()
BUTTON()
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
CMPLE(FP_A,FP_B) - Floating Point Compare if Less Than or Equal
CMPLT(FP_A,FP_B) - Floating Point Compare if Less Than
FLOATADD(FP_X,FP_Y) - Floating Point ADD
FLOATATAN(FP_X,FP_Y) - Floating Point ATAN
FLOATCOS(FP_X,FP_Y) - Floating Point COS
FLOATDIV(FP_X,FP_Y) - Floating Point DIV
FLOATEXP(FP_X,FP_Y) - Floating Point EXP
FLOATLOG(FP_X,FP_Y) - Floating Point LOG
FLOATMUL(FP_X,FP_Y) - Floating Point MUL
FLOATSIN(FP_X,FP_Y) - Floating Point SIN
FLOATSQR(FP_X,FP_Y) - Floating Point SQR
FLOATSUB(FP_X,FP_Y) - Floating Point SUB
FLOATTAN(FP_X,FP_Y) - Floating Point TAN
FLOATTOSTR(FP_A) - Floating Point number to a string
FN()
INSTR([start],Basestring,SearchString)
INT()
JOYSTK()
LEN()
PEEK()
POINT()
RND(x) - Fast random number generator, will generate a value between
1 and x, where x can have a max value of 255
RNDZ(x) - Fast random number generator, will generate a value between
0 and x, where x can have a max value of 255
RNDL(x) - Large random number generator, will generate a value between
1 and x, where x can have a max value of 32767
SGN()
STRTOFLOAT(A$) - Convert a string to a Floating Point variable
SQR()
VAL()
WPEEK()

String Commands it can handle

CHR$()
HEX$()
INKEY$
LEFT$()
MID$()
RIGHT$()
STR$()
STRING$()

Logical operators it can handle

AND
OR
XOR
NOT

Math operators

- +,-,*,/,^, MOD = remainder, DIVR same as / except the result is rounded to the nearest value. For compatibility it accepts \ as integer division (which is the same as /)

New Commands

<u>New SDC commands</u>:

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

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

- is the file number. This must be either a 0 or a 1

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)

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

Points at the 300th byte

n=SDC_GET(1)

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

The next section are 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

GET CURRENT DIRECTORY
A$=SDC_GETCURDIR()

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
12 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 used for audio playback

SDC_PLAY - Playback mono audio samples at 44.75 kHz
SDC_PLAYORC90L - Playback mono audio samples at 44.75 kHz
SDC_PLAYORC90R - Playback mono audio samples at 44.75 kHz
SDC_PLAYORC90S - 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:

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" where the sample MYSAMPLE.RAW is stored on the SD card in your SDC Controller. It is can be created with the FFMPEG command below:

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

New 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.

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**

New Floating Point String conversion commands:

FLOATTOSTR(FP_A)	- Floating Point number to a string
STRTOFLOAT(A$)	- Convert a string to a Floating Point variable

New Floating Point Comparison commands:

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
CMPLE(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
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.

```
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
CMPLE(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_Var1 is > VP_Var6"
```

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 ….
```

You must break it down to:

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

Another example:

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

You must break it down to:

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

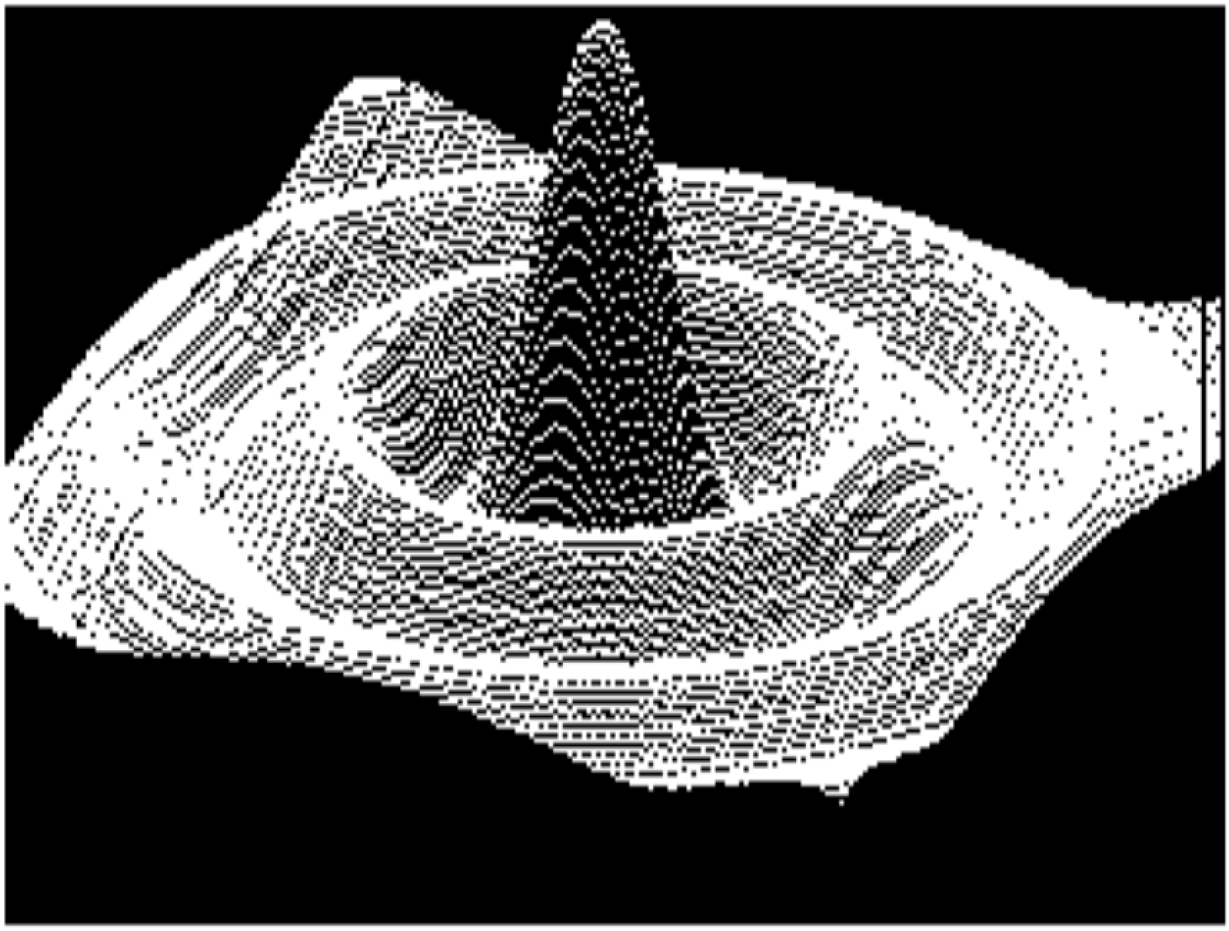
Other Floating Point info and a real world example

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 from working on a CoCo 3 to work on a CoCo 1 & 2



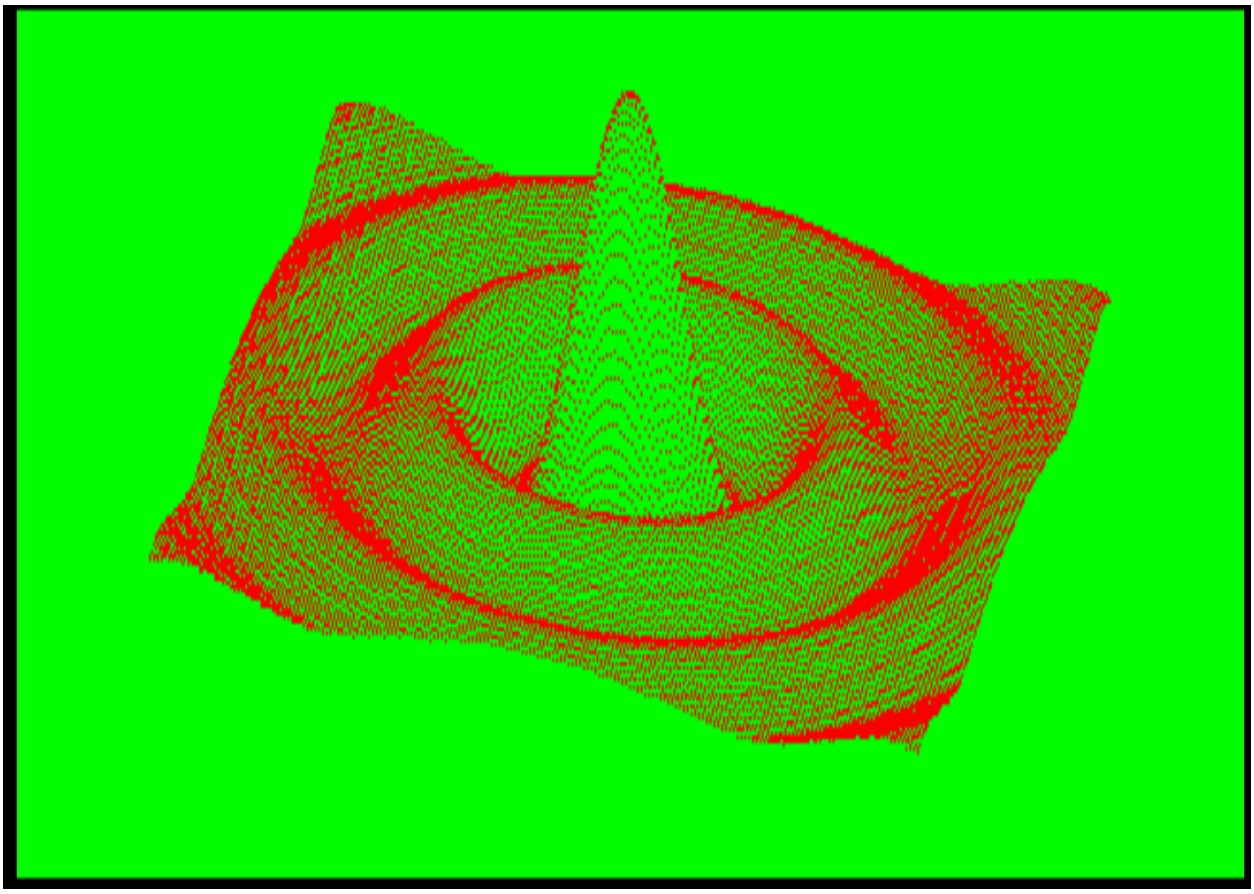
Original program:

```
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:GMODE 16,1:GCLS: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:SET(A,B-F,1)
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
```

New 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

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
Colour - Colour value of 0 to 63

DRAW, GET & PUT commands are not yet available

CoCo 1 & 2 Graphic Modes

GMODE #	Resolution	Colours	Bytes Per 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	3072	Semi graphic-12
8	64 x 192	9	6144	Semi graphic-24
9	64 x 64	4	1024	Full graphic 1-C
10	128 x 64	2	1024	Full graphic 1-R
11	128 x 64	4	2048	Full graphic 2-C
12	128 x 96	2	1536	Full graphic 2-R
13	128 x 96	4	3072	Full graphic 3-C
14	128 x 192	2	3072	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 Per Screen
100	64 x 192	4	3200
101	64 x 200	4	3200
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, * Special mode that repeats the left 4 bytes on the right side of the screen			
122	160 x 200*	2	4000
*(viewable) really 128x192, * Special mode that repeats the left 4 bytes on the right side of the screen			
123	160 x 225*	2	4500
*(viewable) really 128x192, * Special 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
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

GET 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.

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. See here for more info

- 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 sometimes figure out what is causing the problem from the error message and line number given where the compiler found the problem. If you don't use line numbers in your program then the error message won't be able to give you the correct line 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. Scott also wrote the assembly code for the SET and POINT routines used in the semi-graphics modes.

I'd also like to thank others on Discord who inspired me to keep adding new features, including Bruce D. Moore, Erico Monteiro & Pete Willard.