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: 2.15

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.

-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.

-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.

-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.

-pi

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.

-v

Displays the version number of BasTo6809.

-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.

-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

-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
– Use of line numbers is optional
– You can use Labels for sections of code to jump to (case sensitive)
– Variable names can be 30 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.

New commands or features added to BASIC

– Use IF/THEN/ELSE/ELSEIF/ENDIF
– Use SELECT/CASE
– Use WHILE/WEND
– Use DO/WHILE/LOOP
– Use 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. See here for more info
– SDCPLAYORC90L, SDCPLAYORC90R, SDCPLAYORC90S these commands are similar to SDCPLAY except the audio is sent to the Orchestra 90 or COCOFLASH cartridge. See here for more info
– Floating Point commands (special commands to handle floating point calculations and operations. See here for more info

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 my 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 -3bl -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 (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 compilers 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 not matter where it will be loaded into RAM including where the BASIC ROMs are located.

```
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)
DRAW
ELSE
ELSEIF
END
END IF
END SELECT
EVERYCASE
EXEC
EXIT (DO,FOR,WHILE)
FOR/NEXT
GET
GETJOYD (new command which returns values of 0,31 or 63 for both joysticks in both Vert & Horizontal movement) 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
```

GOSUB
GOTO
IF
INPUT
LET
LINE
LOADM
LOCATE - Sets the cursor for text on the PMODE 4 screen LOCATE x,y
Where x is 0 to 31 and y is 0 to 184
LOOP (WHILE/UNTIL)
MOTOR ON/OFF

NEXT
ON GOSUB
ON GOTO
PAINT
PCLS
PLAY
PMODE – Only PMODE 4
POINT
POKE
PRINT – Can’t do PRINT USING, Use PRINT #-3,”Hello” to print on PMODE 4 screen
PRESET
PSET
PUT
READ
RESET
RESTORE
RETURN
SCREEN
SDCPLAY,SDCPLAYORC90L,SDCPLAYORC90R,SDCPLAYORC90S
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
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
FLOATSQRT(FP_X,FP_Y) - Floating Point SQRT
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()
PPOINT()
RND() - Fast random number generator value of 2 to 255
RNDL() - Random number value of 2 to 32767
RNDZ() - Better randomness, but a little slower, value of 2 to 255
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

SDCPLAY, SDCPLAYORC90L, SDCPLAYORC90R and SDCPLAYORC90S

SDCPLAY - Playback an audio file directly stored on the SDC
Usage: SDCPLAY”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

SDCPLAYORCL & SDCPLAYORCR use the same audio format as the regular SDCPLAY 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: SDCPLAYORC90S”MYSAMPLE.RAW” where the sample MYSAMPLE.RAW is stored on the SD card in your SDC Controller. It is 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

New Floating Point Math 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
`FLOATSQRT(FP_Y)` - Floating Point SQRT
`FLOATSUB(FP_X,FP_Y)` - Floating Point SUB
`FLOATTAN(FP_X)` - Floating Point TAN

New Floating Point String conversion command:

`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
`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=FLOATSQRT(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)

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't do complicated equations with floating point math directly. You must do the equations in stages.
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$)`** as

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
`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_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 ….
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 difandaffer'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=FLOATSQRT(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
```

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
- Arrays can only have up to 255 elements per dimension, such as DIM ArrayName(255), but they can also be multidimensional, like ArrayName(10,10,2,3)
- Only supports PMODE 4 graphics and graphic commands
- 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. I also like to thank others on Discord who inspired me to keep adding new features, including Bruce D. Moore, Erico Monteiro.