BY ALLEN C. HUFFMAN

Optimizing Color BASIC,

© 2025 ALLEN C. HUFFMAN. All Rights Reserved.

Reproduction or use, without express written permission from ALLEN C. HUFFMAN, of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, ALLEN C. HUFFMAN assumes no liability resulting from any errors or omissions in this manual, or from the use of the information contained herein.

TRS-80 Extended Color BASIC System Software; @ 1984 Tandy Corporation and Microsoft.

All Rights Reserved.
The system software in the Color Computer is retained in a read-only memory (ROM) format. All portions of this system software, whether in the ROM format or other source code form format, and the ROM circuitry, are copyrighted and are the proprietary and trade secret information of Tandy Corporation and Microsoft. Use, reproduction, or publication of any portion of this material without the prior written authorization by Tandy Corporation is strictly prohibited.

Book Layout & Design by Carlos. A. Camacho

10 9 8 7 6 5 4 3 2 1

Dedicated to the many CoCo users around the world that have contributed to this series on Allen's blog.

https://subethasoftware.com/

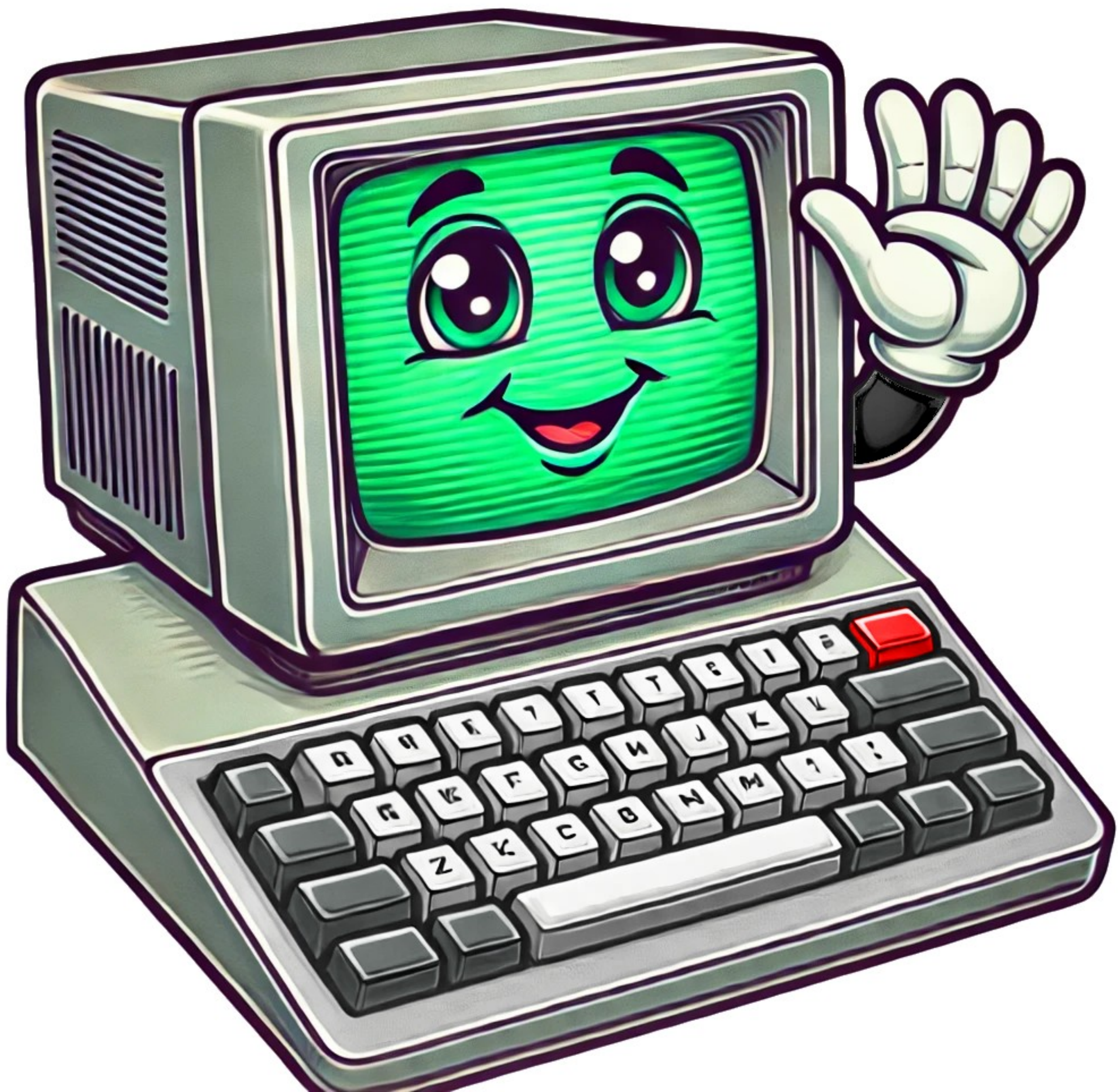


Table of Contents

Part 1 ... 1
- Remove all spaces ... 1
- Pack/combine lines where possible ... 1
- Remove all REMs ... 1
- Renumber by 1 ... 2
- Removing NEXT variables ... 2

Part 2 ... 6
- Variable Placement ... 6

Part 3 ... 14
- INSTR ... 19
- POKE ... 21

Part 4 ... 27
- INSTR and GOTO/GOSUB ... 27
- But wait, there’s more! ... 31

Part 5 ... 39
- HEX versus DECimal Numbers ... 39

Part 6 ... 44
- Size Matters. Or Space Matters. You decide ... 44
- Elementary, my dear DATA ... 45
- Base 10 Numbers ... 47
- Hexadecimal Base-16 Numbers ... 48
- String HEX Numbers ... 50
- Bonus Data ... 50

Part 7 ... 55
- GOSUB Revisted ... 55
- Optimizing Color Basic - ON GOTO vs ON GOSUB ... 55
- A few additional REMarks ... 60

Part 8 ... 62
- Arrays and Variable Length ... 62
- Variable Length ... 64

Part 9 ... 69
- “.” versus “0” ... 69
- Fake FOR ... 69

Part 1

Remove all spaces

The less bytes to parse, the smaller and faster it will run. It has to be smart enough to know when spaces are required, such as "`FORA=1TOQ STEP1`" when a variable needs a space after it before another keyword.

Pack/combine lines where possible

Any line called by a `GOTO` had to remain at the start, by following lines would be combined up to the max size of a BASIC line. Only when it had to be broken by logic (`ELSE`, etc.) would it not be combined. For example:

```
10 FOR I=1 TO 100
20 GOSUB 50
30 NEXT I
40 END
50 REM PRINT SOMETHING
60 PRINT "HELLO WORLD!
70 RETURN
```

…would end up as:

```
10 FOR I=1 TO 100: GOSUB 60:NEXT I: END
60 PRINT"HELLO WORLD!":RETURN
```

Remove all REMs

Obvious. I *think* his program would adjust a `GOTO`/`GOSUB` that went to a `REM` line to go to the next line after it, and remove the `REM`:

```
10 GOSUB 1000
20 END
1000 REM PRINT SOMETHING
1010 PRINT "HELLO"
1020 RETURN
```

…would become like:

```
10 GOSUB1010:END
1010 PRINT"HELLO":RETURN
```

NOTE: Even without packing, I learned not to `GOTO` or `GOSUB` to a `REM` since it required the interpreter to parse through the line. I would GO to the first line of code after the `REM`:

```
10 GOSUB 1010
...
1000 REM MY ROUTINE
1010 PRINT "HERE IS WHERE IT STARTS"
```

Everything you do like that saves a bit of parsing time since it doesn't have to start parsing 1000 and look up a token then realize it can ignore the rest of the line.

Not bad so far—but that's not all!

The BASIC input buffer is some maximum number of bytes (250?). When you type a line to max, it stops you from typing more. When you hit ENTER, it is tokenized and could be much smaller. For instance, "`PRINT`" turns into a one-byte token instead of the five characters. Carl's program would combine and pack the tokens up to the max line size. Thus, it created lines that BASIC could run which were IMPOSSIBLE to enter on the keyboard. If I recall, you could `LIST` and they would print (?) but if you did an `EDIT` you were done.

Renumber by 1

`GOSUB2000` would take up one byte for the token, then four bytes for the line number. If you renumber by 1s, it might make that function be `GOSUB582` and save a byte. Multiply that by every use of `GOTO`/`GOSUB`/`ON GOTO`/etc. and you save a bit.

Even without one of these compressors, try a before/after just doing a `RENUM 1,1,1`. I recently posted an article with a short (27 line) word wrap routine in BASIC. Doing this `RENUM` saved 7 bytes.

Removing NEXT variables

I am unsure if this is just something I knew, or if his program also did it. If you use `FOR`/`NEXT` and it is used normally, you don't need the `NEXT` variables like this:

```
10 FOR D=0 TO 3 'DRIVES
20 FOR T=0 TO 34 'TRACKS
30 FOR S=1 TO 18 'SECTORS
40 DSKI$ D,T,S,S1$,S2$
50 NEXT S
60 NEXT T
70 NEXT D
```

The `NEXTs` could also be

```
50 NEXT S,T,D
```

Or

```
50 NEXT:NEXT:NEXT
```

Ignoring spaces, "`NEXT:NEXT:NEXT`" takes up one less byte than "`NEXTS,T,D`" (`NEXT` is turned into a token, so it is TOKEN COLON TOKEN COLON TOKEN versus TOKEN BYTEVAR COMMA BYTEVAR COMMA BYTEBAR".

This takes up less memory, but there is a speed difference:

```
10 T=TIMER
20 FOR I=1 TO 10000
30 REM DO NOTHING
40 NEXT I
50 PRINT TIMER-T
```

Run this and it shows something in the 1181 range (speed varies based on other things BASIC does; pound on keys while it runs and it takes more time, for example). Change the "`NEXT I`" to "`NEXT`" and it shows something in the range of 1025. The more `FOR` variables, the more the savings, too.

```
10 TM=TIMER
20 FOR D=0 TO 3
30 FOR T=0 TO 34
40 FOR S=1 TO 18
50 REM DO SOMETHING
60 NEXT S
70 NEXT T
80 NEXT D
90 PRINT TIMER-TM
```

…shows around 345. Changing it to:

```
60 NEXT S,T,D
```

…changes it to around 336 – slightly smaller and faster since it is not parsing three different lines.

```
60 NEXT:NEXT:NEXT
```

…changes it to around 293! One byte smaller and slightly faster.

AND THERE'S MORE! But this post is already too long.

What I think would be cool is an actual BASIC OPTIMIZER that could do some smart things to programs, much like we do with the C language optimizers for ASM and C. For example, noticing stupid things:

```
A$ = B$+"."+"."+C$
```

to

```
A$ = B$+".."+C$
```

And removing `NEXT` variables, or doing minor restructuring based on what it knows about the interpreter.

I suppose, back in the 80s, processors were so slow and memory so limited that doing such an optimizer might not be practical. But today, imagine how you could optimize a BASIC on a modern computer (just like we have 6809 cross-hosted compilers that compile and link in the blink of an eye).

Maybe this has already been done for some other favor of BASIC? Hmmm. I may have to do some Googling.

Comment 1

You've pretty well covered the things that are specific to the peculiarities of the BASIC interpreter, though come to think of it here's something you could do: read the source and count how many times the variables are used. Give the 26 most-used ones one-letter names.

Does Color BASIC have DEF FN() = ? If it does, it might be worth looking for expression schemata that appear several times, generating the appropriate DEF FNx, and replacing the occurrences of the expression with calls. You'd want to test to see whether it really saves space.

Past that,. you're talking real live optimization, with real live data and control flow analysis. Things like:

See how many places GOSUB to a particular subroutine. If the answer is "one", inline it.

Parse expressions and do constant folding (like the ":" + ":") and other optimizations on them.

Strength reduction could be tricky–you'd have to flatten arrays into one dimension to let you do strength reduction in the language.

Another tricky part: that code you're optimizing may be some carefully-tweaked delay loop, so you'll have to know when not to optimize..Probably that means programmer marking:

```
100 REM NOOPT CODE TO BE LEFT ALONE
200 REM OPT
```

Speaking of REM, I presume you'd replace, um, is it PRINT or INPUT with ? and REM with '

Ah yes! Once character variable names. I completely forgot to mention that. I am not sure if Carl England's program touched variable names, but that would make sense. One thing I did (after reading about it somewhere) even back in 1983 was declare variables at the start with a "DIM A$,A,B,C" which caused it to per-allocate a spot for them. Perhaps, if BASIC does not sort them as they are created, doing such a DIM makes them in that order, so the most often used ones could be declared first so it would not need to search through 50 variables looking for "I".

DEF FN does exist in Extended Color BASIC. I am going to have to read up on that. I recall it existing, but do not remember what it is for.

In CoCo BASIC, REM is one token, and ' turns in to :REM and takes two tokens, so doing this:

```
10 REM THIS IS A NOTE VERSUS
10 ' THIS IS A NOTE
```

…it is better to use the first, "REM(SPACE)" being two bytes, and "(COLON)REM(SPACE)" being three. BUT, if you left out the space:

```
10 REM THIS IS A NOTE
10 REM 'THIS IS A NOTE
```

…they are the same.

Comment 2

It looks like I have a few more updates to do on this topic — thanks!

Great notes, James! I understand why no one cared or bothered doing this for CoCo BASIC, but it would be a fun experiment just to see how much of a difference it can make.

Wow, DEFFN looks quite useful! I do not recall ever using it. In a program I am playing with, I turn an X and Y in to a memory location based on the starting address of screen memory:

```
10 P=1024+Y*32+X:POKE P,42
```

I converted that to:

```
DEF FNP(X Y)=1024+Y*32+X
10 P=FNP(X Y):POKE P,42
```

That could be a good code space savings if it is used many times. Speed tests with one parameter seemed a tiny bit faster, but with two, it was slower. Trade-offs.

Excellent.

Comment 3

Drats. DEFFN in CoCo BASIC claims to handle multiple variables, but it was not implemented. It works with one, but with multiple, it is just using one parameter and then reading the global variables used at DF() time. That's a workaround, but not much different than a GOSUB.

Comment 4

Oh, yeah… Someone long ago did and wrote up a paper on source-to-source optimizations in FORTRAN. Perhaps it would give you some ideas.

Comment 5

Another Color BASIC optimization I recall was to put subroutines you call often at low line numbers because when you call GOSUB it starts from the first line number and scans up until it finds the line number that matches the target.

Comment 6

You are correct! Less search time. Though I think it searches forward if it can, so if there were a bunch of subroutines up front there might be some instances where having one after the call could be found faster. Interesting!

Comment 7

Oh yeah. If anyone cares to experiment with modifying the BASIC interpreter, it might be fun to make the symbol table "adaptive". When you find a variable in the symbol table, if it's not the first one, swap it with its predecessor. The idea is that the more frequently looked up symbols migrate towards the front and thus are more quickly found. The question is whether the migration improves things enough to make up for the swapping.

Comment 8

I will have to do some tests. At the very least, perhaps doing a "DIM" at the top defining all the variables would put them in that order, so you could put the most used up there first. I'll do some quick timing tests…

Part 2

Variable Placement

Last year, I posted an article dealing with Optimizing Color BASIC. In it, I covered a variety of techniques that could be done to speed up a BASIC program. While this was specifically written about the Microsoft Color BASIC on the Radio Shack Color Computers, I expect it may also apply to similar BASICs on other systems.

James J. left an interesting comment:

> *Oh yeah. If anyone cares to experiment with modifying the BASIC interpreter, it might be fun to make the symbol table "adaptive". When you find a variable in the symbol table, if it's not the first one, swap it with its predecessor. The idea is that the more frequently looked up symbols migrate towards the front and thus are more quickly found. The question is whether the migration improves things enough to make up for the swapping. – James J.*

This got me curious as to how much of a difference this would make, so I did a little experiment.

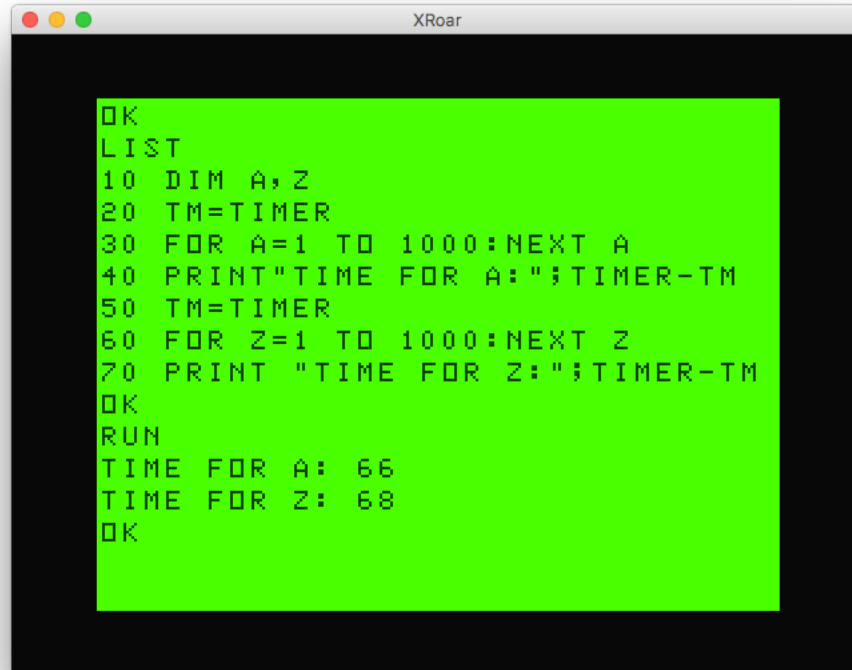
In this Microsoft BASIC, variables get created when you first use them. Early on, I learned a tip that you could define all your variables at the start of your program and get that out of the way before your actual code begins. You can do this with the `DIM` statement:

```
DIM A,B,A$,B$
```

Originally, I thought `DIM` was only used to define an array, such as `DIM A$(10)`.

I decided to use this to test how much of a difference variable placement makes. Variables defined first would be found quicker when you access them. Variables defined much later would take more time to find since the interpreter has to walk through all of them looking for a match.

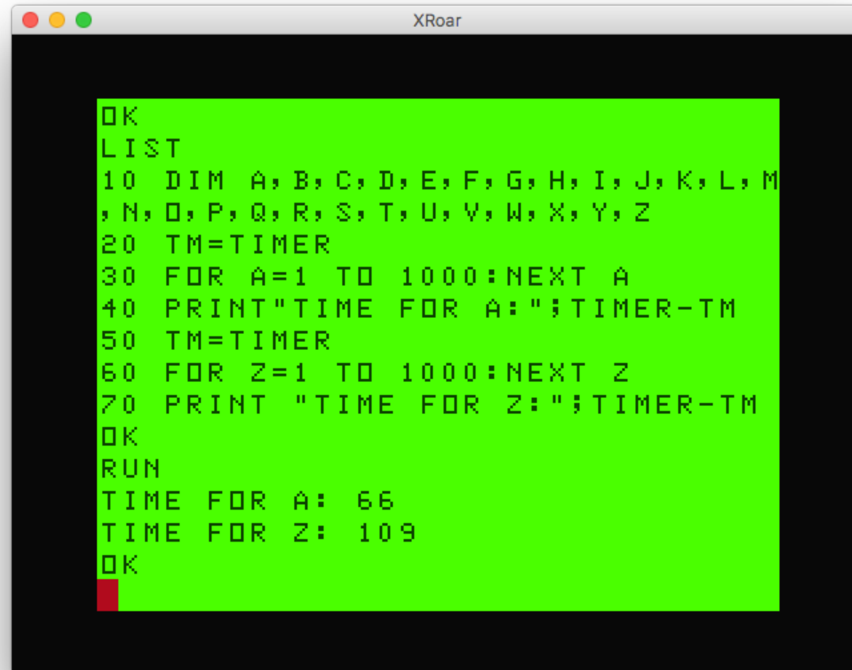
Using the XRoar CoCo/Dragon emulator, I wrote a simple test program that timed two `FOR`/`NEXT` loops using two different variables. It looks like this:



In BASIC, variables defined earlier are faster.

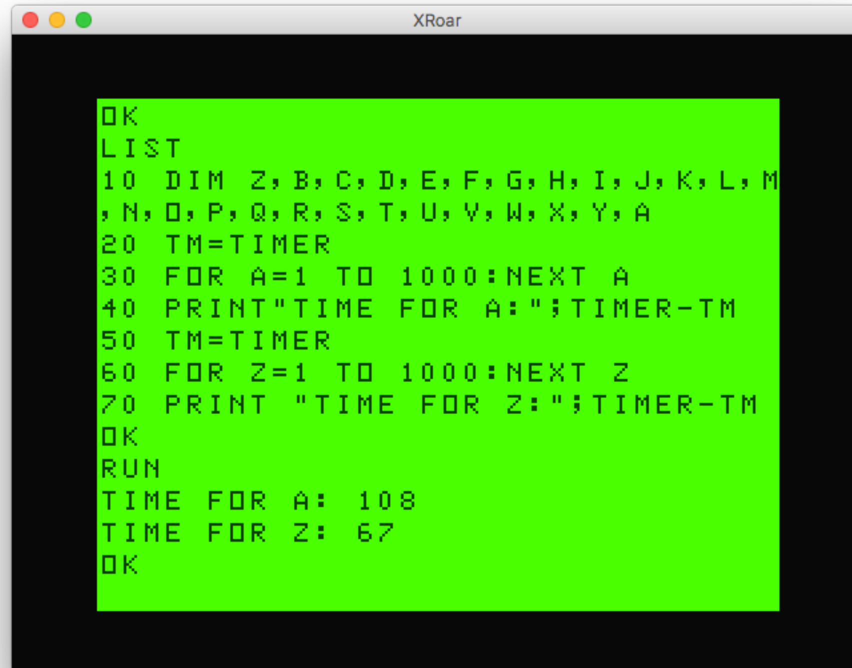
As you can see, with just two variables, A and Z, there wasn't much difference between the time it takes to use them in a small `FOR`/`NEXT` loop. I expect if the loop time was much later, you'd see more and more differences.

But what if there were more variables? I changed line 10 to define 26 different variables (A through Z) then ran the same test:



In BASIC, variables defined last take longer to find, so they are slower.

Now we see quite a bit of difference between using A and using Z. If I knew Z was something I would be using the most, I might define it at the start of the `DIM`. I did another test, where I defined Z first, and A last:



In BASIC, define the most-used variables first to speed things up.

As expected, now the Z variable is faster than A.

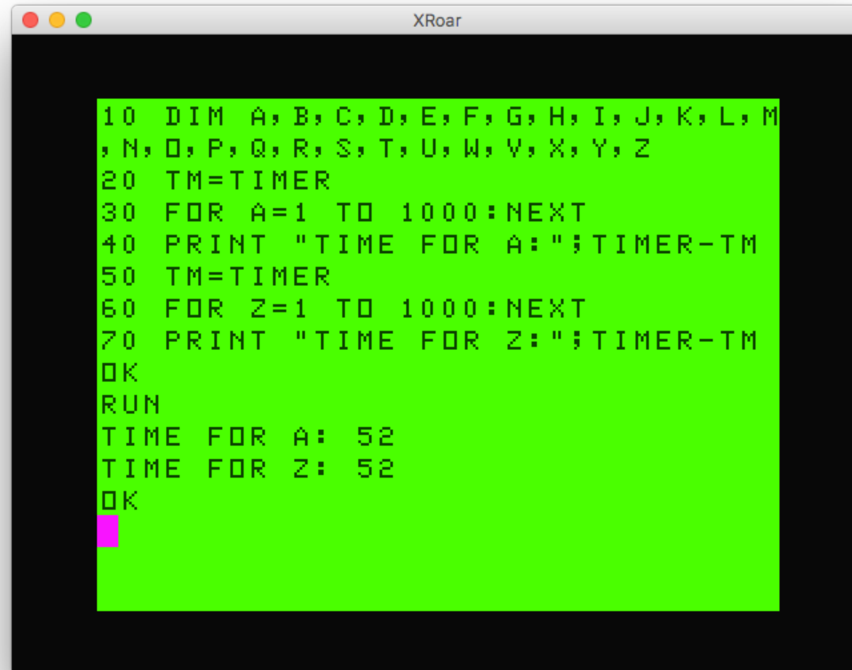
Every time BASIC has to access a variable, it makes a linear (I assume) search through all the variables looking for a match.

Side Note: There is an excellent Super/Disk/Extended/Color Basic Unraveled book set which contains fully commented disassembles of the ROMs. I could easily stop assuming and actually know if I was willing to take a few minutes to consult these books.

However, when I first posted these results to the Facebook CoCo group, James responded there:

> Didn't realize it made that much difference–doesn't the interpreter's FOR loop stack remember the symbol table entry for the control variable?

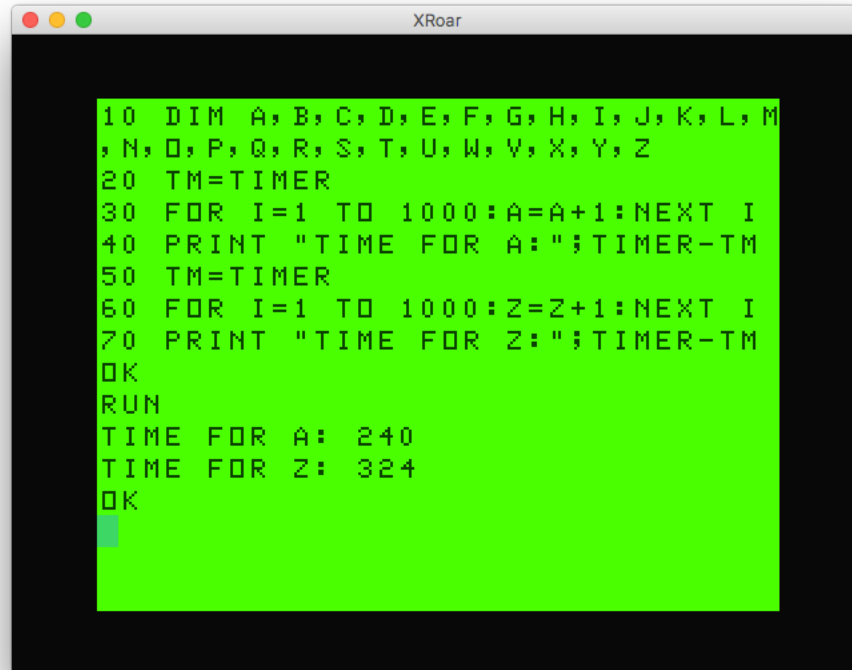
Indeed, this does seem to be a bad test. `FOR`/`NEXT` does not need the variable after the `NEXT`. If you omit the variable (just using `NEXT` by itself), it does not need to do this lookup and both get faster:



NEXT without a variable is faster.

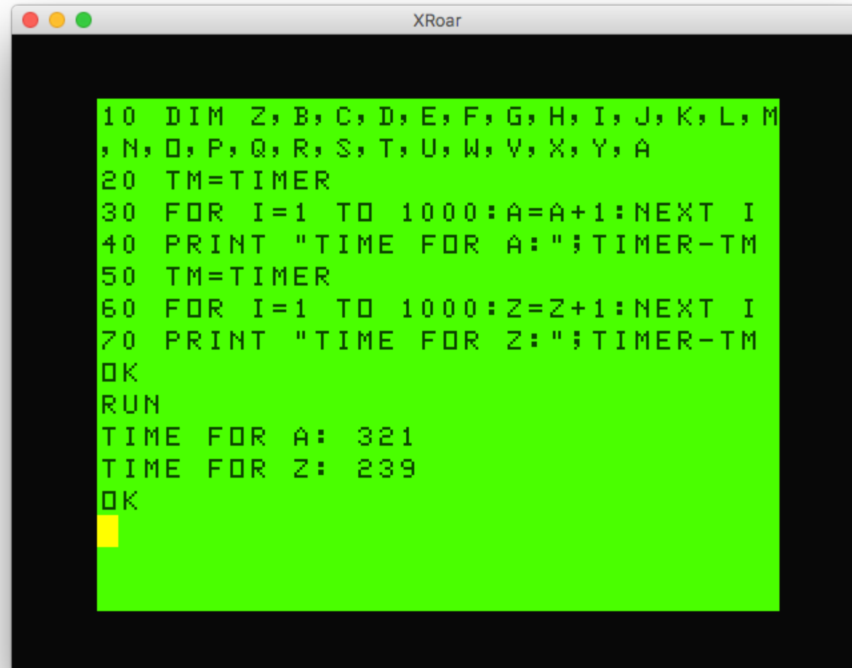
I guess I need a better test.

How about using the variable directly, such as simple addition?



Variable addition is slower for later variables.

Z, being defined at the end, is slower. And if we reverse that (see line 10, defining Z first), Z becomes faster:



Variable addition is faster for earlier variables.

You can speed up programs by defining often-used variables earlier.

James' suggestion about modifying the interpreter to do this automatically is a very interesting idea. If it continually did it, the program would adapt based on current usage. If it entered a subroutine that did a bunch of work, those variables would become faster, then when it exited and went back to other code, those variables would become faster.

I do not know if the BASIC language lasted long enough to ever evolve to this level, but it sure would be fun to apply these techniques to the old 8-bit machines and see how much better (er, faster) BASIC could become.

Thanks for the comment, James!

Comment 1

As a point of information on modifying the interpreter to somehow shuffle more frequently used variables to the front of the variable table, that is actually harder than it sounds. It would require storing extra information (access count, for instance). On a system that is already limited in memory, that would probably be intolerable. I could see how you could then use the access count to swap a variable with its preceding one if its access count is higher without too much of a performance hit. However, that tiny extra time would have to apply to every single variable access. Also, you would have to have a periodic sweep that would reset the counters somehow to prevent overflows from creating table churn.

Probably a better solution would be to use an ordered list of variables which would allow using a binary search to find

variable entries. Doing that would give O(log n) for lookup times and, so, would be a lot more predictable with a substantially better worst case. That would make creating a variable entry slower but that only happens once. It also wouldn't require any extra storage space compared to what is currently used. To make things even more fun, as long as the list is always kept ordered, adding a new variable would be at worst O(n).

What if you simply swapped places with the variable before you? No counts needed, but items accessed more often would work their way to the top.

Comment 2

Then you're going to be swapping variable table entries for every variable access that isn't the first in the table. I haven't bench marked it, but I think that's going to have a fairly substantial net slowdown in the average case where a program accesses a few variables randomly rather than just the same variable over and over.

Swapping 7 bytes takes CPU cycles and it will add up. It takes 16 instructions to do the swap (two loads and two stores per swap, maximum two bytes at a time) barring some truly diabolical optimization scheme, and that's on top of the two instructions needed to decide if the variable entry should be swapped. That will add up substantially in a non-trivial program.

(7 bytes comes from two bytes for the variable name and 5 bytes for the value or string descriptor.)

I think the overall best case on average would be an ordered variable list and a binary search. Potentially slow variable creation but quite fast searches. (7 comparisons on a variable table with 128 entries, for instance.)

Obviously, bubbling variables to the front of the line (your described solution) will be faster in some cases. Even the existing system is faster in some cases. The ordered list with a binary search is more predictable, though, and not prone to accidentally creating a worst-case.

Part 3

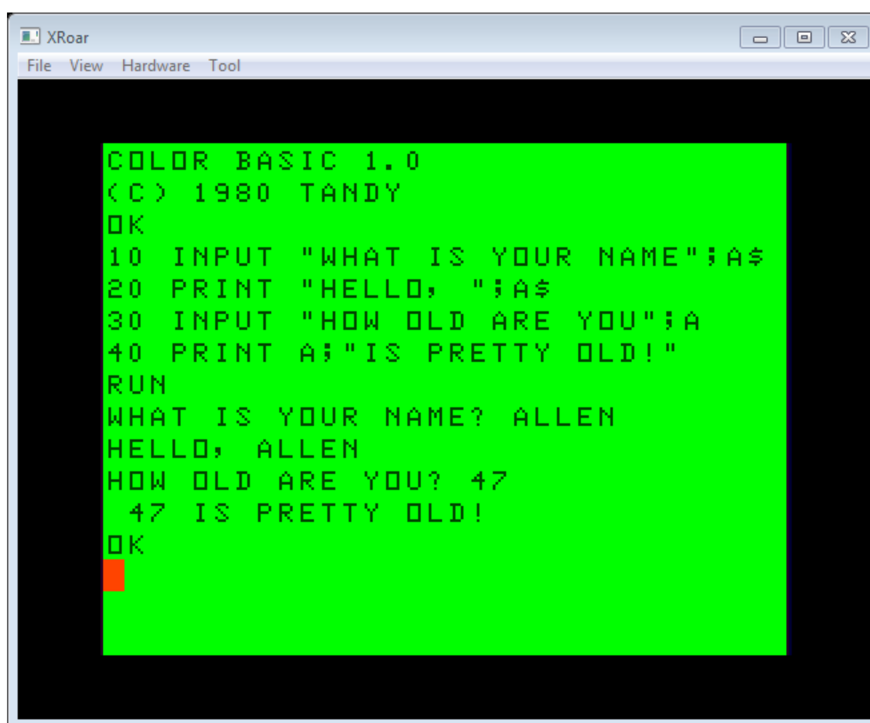
Since I am right in the middle of a multi-part article on interfacing assembly with BASIC, now is a great time to discuss something completely different.

INPUT, INKEY, INSTR, AND POKE

The reason I do this now is because it is going to tie it in with the next part of the assembly article. Since I have been discussing using assembly to speed things up, it is a good time to address a few more things that can be done to speed up BASIC before resorting to 6809 code. Since BASIC will be the weakest link, we should try to make it as strong weak link.

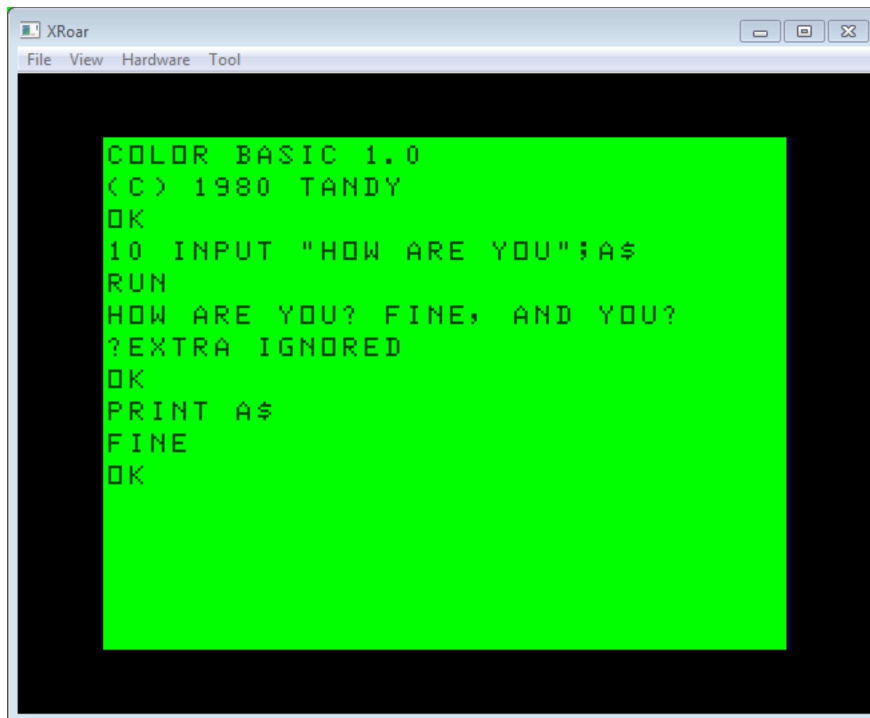
In 1980, Color BASIC offered a simple way to input a string or number:

```
10 INPUT "WHAT IS YOUR NAME";A$
20 PRINT "HELLO, ";A$
30 INPUT "HOW OLD ARE YOU";A
40 PRINT A;"IS PRETTY OLD."
```

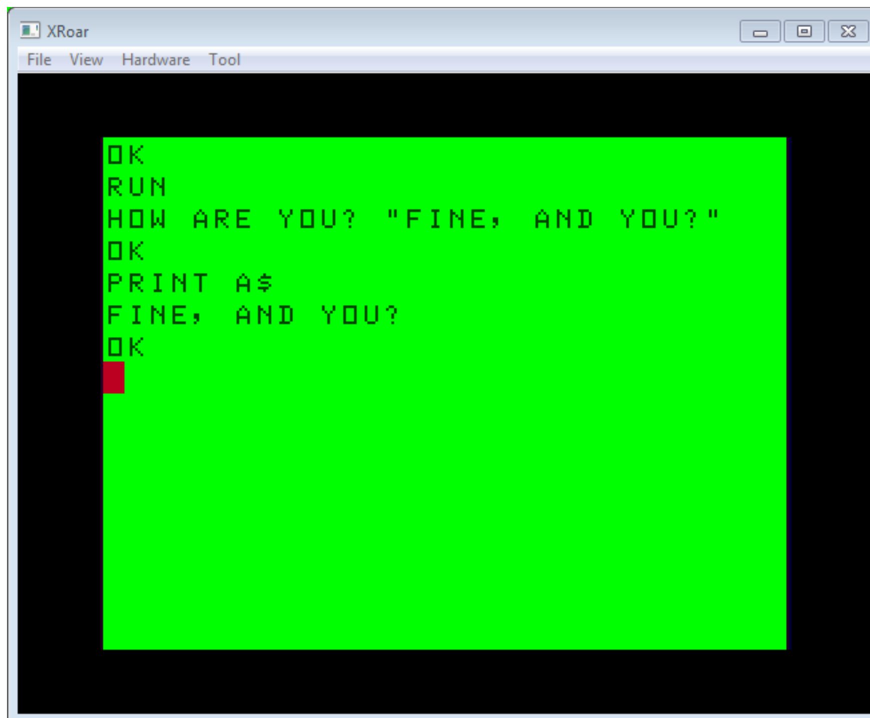


My First Input

The original `INPUT` command was a very simple way to get data in to a program, but it was quite limited. It didn't allow for string input containing commas, for instance, unless you typed it in quotes:



INPUT hates commas.



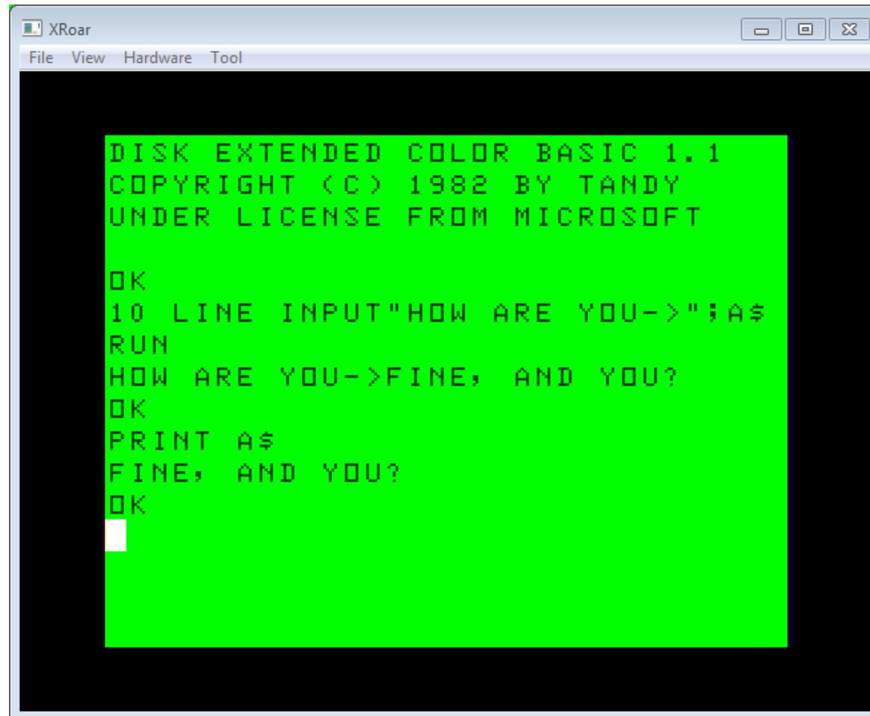
INPUT likes quotes.

`INPUT` also prints the question mark prompt.

LINE INPUT

When Extended Color BASIC was introduced, it brought many new features including the `LINE INPUT` command.

This command did not force the question mark prompt, and would accept commas without quoting it:



LINE INPUT likes everything.

If you were trying to write a text based program (or a text adventure game), `INPUT` or `LINE INPUT` would be fine.

INKEY$

For times when you just wanted to get one character, without requiring the characters to be echoed as the user types them, and without requiring the user to press ENTER, there was `INKEY$`.

`INKEY$` returns whatever key is being pressed, or nothing ("") if no key is ready.

```
10 PRINT "PRESS ANY KEY TO CONTINUE..."
20 IF INKEY$="" THEN 20
30 PRINT "THANK YOU."
```

It can also be used with a variable:

```
10 PRINT "ARE YOU READY? ";
20 A$=INKEY$:IF A$="" THEN 20
30 IF A$="Y" THEN PRINT "GOOD!" ELSE PRINT "BAD."
```

This is the method we might use for a keyboard-controlled BASIC video game. For instance, if we want to read the arrow keys (up, down, left and right), each one of those keys generates an ASCII character when pressed:

```
UP          - CHR$(94)  - ↑ CHARACTER
DOWN        - CHR$(10)  - LINE FEED
LEFT        - CHR$(8)   - BACKSPACE
RIGHT       - CHR$(9)   - TAB
```

Knowing this, we can detect arrow keys using INKEY$:

```
10 CLS :P=256+16
20 PRINT@P,"*";
30 A$=INKEY$:IF A$="" THEN 30
40 IF A$=CHR$(94) AND P>31 THEN P=P-32
50 IF A$=CHR$(10) AND P<479 THEN P=P+32
60 IF A$=CHR$(8) AND P>0 THEN P=P-1
70 IF A$=CHR$(9) AND P<510 THEN P=P+1
80 GOTO 20
```

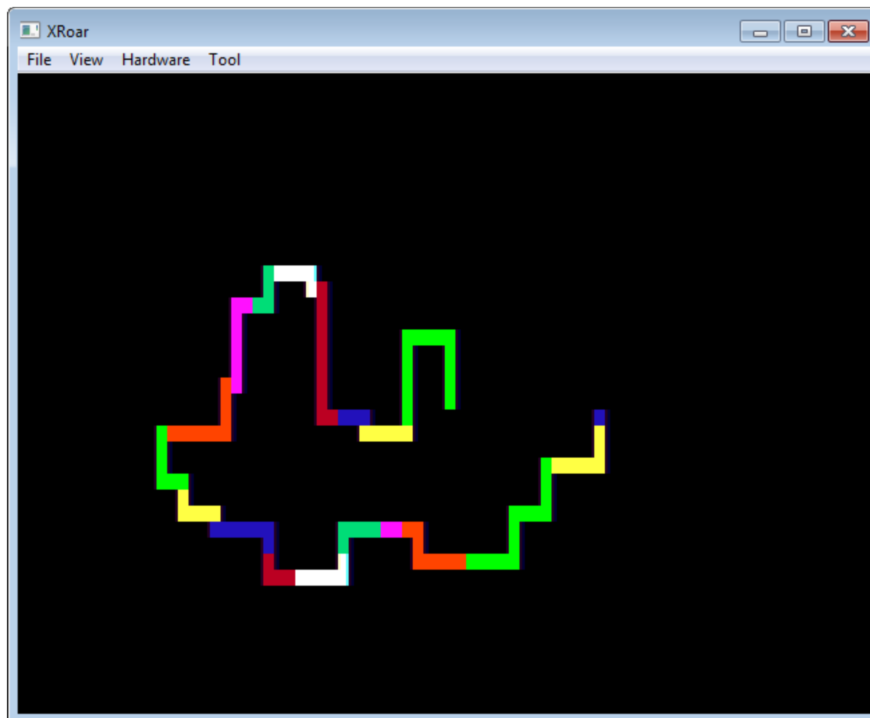
The above program uses PRINT@ to print an asterisk ("*") in the middle of the screen. Then, it waits until a key is pressed (line 30). Once a key is pressed, it looks at which key it was (up, down, left or right) and then will move the position of the asterisk (assuming it's not going off the end of the screen).

Side Note: The CoCo's text screen is 32×16 (512 characters). PRINT@ can print at 0-511, but if you print to 511 (the bottom right location), the screen will scroll up. I have adjusted this code to disallow moving the asterisk to that location.

You now have a really crappy text drawing program. To make it less crappy, you could check for other keys to change the character that is being drawn, or make it use simple color graphics:

```
10 CLS0:X=32:Y=16:C=0
20 SET(X,Y,C)
30 A$=INKEY$:IF A$="" THEN 30
40 IF A$=CHR$(94) AND Y>0 THEN Y=Y-1
50 IF A$=CHR$(10) AND Y<31 THEN Y=Y+1
60 IF A$=CHR$(8) AND X>0 THEN X=X-1
70 IF A$=CHR$(9) AND X<63 THEN X=X+1
80 IF A$="C" THEN C=C+1:IF C>8 THEN C=0
90 GOTO 20
```

That program uses the primitive SET command to draw in beautiful 64×32 resolution with eight colors. The arrow keys move the pixel, and pressing C toggles through the colors. Spiffy!



Color graphics from 1980!

Instead of using "C" to just cycle through the colors, you could check the character returned and see if it was between "0" and "8" and use that value to set the color (`0=RESET` pixel, `1-8=SET` pixel to color).

```
10 CLS0:X=32:Y=16:C=1
20 IF C=0 THEN RESET(X,Y) ELSE SET(X,Y,C)
30 A$=INKEY$:IF A$="" THEN 30
40 IF A$=CHR$(94) AND Y>0 THEN Y=Y-1
50 IF A$=CHR$(10) AND Y<31 THEN Y=Y+1
60 IF A$=CHR$(8) AND X>0 THEN X=X-1
70 IF A$=CHR$(9) AND X<63 THEN X=X+1
80 IF A$=>"0" AND A$="8" THEN C=ASC(A$)-48
90 GOTO 20
```

Now that we have refreshed our 1980 BASIC programming, let's look at lines 40-70 which are used to determine which key has been pressed.

If we are going to be reading the keyboard over and over for an action game, doing so with a bunch of `IF`/`THEN` statements is not very efficient. Lets do some tests to find out how not very efficient it is.

For our example, we would be using `INKEY$` to read a keypress, then `GOSUB`ing to four different subroutines to handle up, down, left and right actions. To see how fast this is, we will once again use the `TIMER` command and do our test 1000 times. We'll skip doing the actual `INKEY$` for now, and hard code a keypress. Since we will be checking for keys in the order of up, down, left then right, we will simulate pressing last key check, right, to get the worst possible condition.

Here is version 1 that does a brute-force check using `IF`/`THEN`/`ELSE`.

```
0 REM KEYBD1.BAS
10 TM=TIMER:FORA=1TO1000
15 A$=CHR$(9)
20 REM A$=INKEY$:IFA$=""THEN20
30
IFA$=CHR$(94)THENGOSUB100ELSEIFA$=CHR$(10)THENGOSUB200ELSEIF
A$=CHR$(8)THENGOSUB300ELSEIFA$=CHR$(9)THENGOSUB400
50 NEXT:PRINT TIMER-TM
60 END
100 RETURN
200 RETURN
300 RETURN
400 RETURN
```

When I run this in the XRoar emulator, I get back **1821**. That is now our benchmark to beat.

INSTR

Rather than doing a bunch of `IF`/`THEN`s, if we are using Extended Color BASIC, there is the `INSTR` command. It will take a string and a pattern, and return the position of that pattern in the string. For example:

```
PRINT INSTR("CAT DOG RAT", "DOG")
```

If you run this line, it will print 5. The string "DOG" appears in "CAT DOG RAT" starting at position 5. You can use `INSTR` to parse single characters, too:

```
PRINT INSTR("ABCDEFGHIJ", "F")
```

This will print 6, because "F" is found in the search string starting at position 6.

If the search string is not found, it returns 0. Using this, you can parse a string containing all the possible key press options, and turn them in to a number. You could then use that number in an `ON GOTO`/`GOSUB` statement, like this:

```
0 REM KEYBD2.BAS
10 A$=INKEY$:IF A$="" THEN 10
20 A=INSTR("ABCD",A$)
30 IF A=0 THEN 10
40 ON A GOSUB 100,200,300,400
50 GOTO 10
100    PRINT    "A   WAS  PRESSED":RETURN
200    PRINT    "B   WAS  PRESSED":RETURN
300    PRINT    "C   WAS  PRESSED":RETURN
400    PRINT    "D   WAS  PRESSED":RETURN
```

A long line of four `IF`/`THEN`/`ELSE` statements is now replaced by `INSTR` and `ON GOTO`/`GOSUB`.

Let's rewrite our test program slightly, this time using `INSTR`:

```
10 TM=TIMER:FORA=1TO1000
15 A$=CHR$(9)
20 REM A$=INKEY$:IFA$=""THEN20
30 LN=INSTR(CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9),A$)
35 IF LN=0 THEN 20
40 ONLN GOSUB100,200,300,400
50 NEXT:PRINT TIMER-TM
60 END
100 RETURN
200 RETURN
300 RETURN
400 RETURN
```

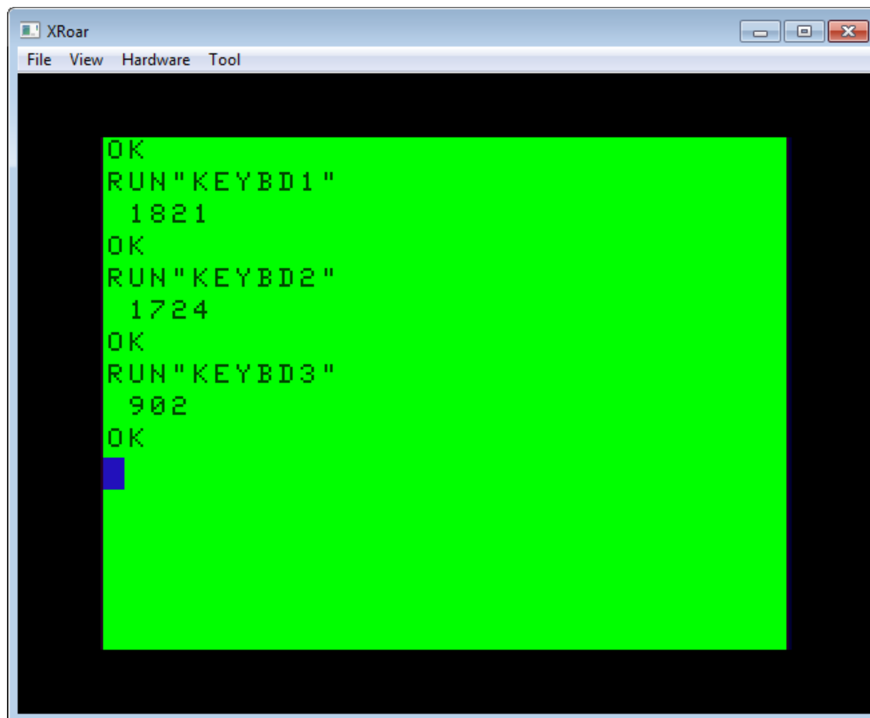
Running this gives me **1724**. We are now *slightly* faster.

We can do better.

One of the reasons this version is so slow is line 30. Every time that line is processed, BASIC has to dynamically build a string containing the four target characters — `CHR$(94)`, `CHR$(10)`, `CHR$(8)` and `CHR$(9)`. String manipulation in BASIC is slow, and we really don't need to do it every time. Instead, let's try a version 3 where we create a string containing those characters at the start, and just use the string later:

```
0 REM KEYBD3.BAS
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 TM=TIMER:FORA=1TO1000
15 A$=CHR$(9)
20 REM A$=INKEY$:IFA$=""THEN20
30 LN=INSTR(KB$,A$)
35 IF LN=0 THEN 20
40 ONLN GOSUB100,200,300,400
50 NEXT:PRINT TIMER-TM
60 END
100 RETURN
200 RETURN
300 RETURN
400 RETURN
```

Running this gives me **902**! It appears to be **twice as fast** as the original `IF`/`THEN` version!



Speed comparisons…

Now we have a much faster way to handle the arrow keys. Let's go back to the original program and update it:

```
0 REM INKEY3.BAS
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=256+16
20 PRINT@P,"*";
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$)
50 ONLN GOSUB100,200,300,400
60 GOTO 20
100 IF P>31 THEN P=P-32
110 RETURN
200 IF P<479 THEN P=P+32:RETURN
210 RETURN
300 IF P>0 THEN P=P-1
310 RETURN
400 IF P<510 THEN P=P+1
410 RETURN
```

Side Note: Using `GOSUB`/`RETURN` may be slower than using `GOTO`, but that will be the subject of another installment.

Now that we have a faster keyboard input routine, let's do one more thing to try to speed it up.

POKE

We are currently using PRINT@ to print a character on the screen in positions 0-510 (remember, we can't print to the bottom right position because that will make the screen scroll). Instead of using PRINT, we can also use POKE to put a byte directly into screen memory:

```
POKE LOCATION,VALUE
```

Location is an address in the up-to-64K memory space (0-65535) and value is an 8-bit value (0-255).

Let's see if it's faster.

First, PRINT@ wants positions 0-511, and POKE wants an actual memory address. The 32-column screen is located from 1024-1535 in memory, so PRINT@0 is like POKE 1024. PRINT@511 is like POKE 1535. Let's make some changes:

```
0 REM INKEY4.BAS
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=1024+256+16
20 POKE P,106
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$)
50 ONLN GOSUB100,200,300,400
60 GOTO 20
100 IF P>1024+31 THEN P=P-32
110 RETURN
200 IF P<1024+479 THEN P=P+32:RETURN
210 RETURN
300 IF P>1024+0 THEN P=P-1
310 RETURN
400 IF P<1024+511 THEN P=P+1
410 RETURN
```

WARNING: While PRINT@ is safe (a bad value just generates an error), POKE is dangerous! If you POKE the wrong value, you could crash the computer. Instead of POKEing a character on the screen, you could accidentally POKE to memory that could crash the system.

This program will behave identically to the original, BUT since we are using POKE, we can now go all the way to the bottom right of the screen. That is just one of the reasons we might use POKE over PRINT@.

But is it faster, or slower? Let's find out…

```
10 TM=TIMER:FORA=1TO1000
20 PRINT@0,"*";
30 NEXT:PRINT TIMER-TM
```

…versus…

```
10 TM=TIMER:FORA=1TO1000
20 POKE1024,106
30 NEXT:PRINT TIMER-TM
```

The PRINT@ version shows **259**, and the POKE version shows **655**. POKE appears to be significantly slower. Some reasons could be:

1) POKE has to translate four digits (1024) instead of just one (0) so that's longer to parse.

2) POKE has to also translate the value (106) where PRINT can probably just jump to the string that is in the quotes.

Let's try to test this… By giving PRINT@ a three-digit number, 510, it slows down from 259 to **424**. Parsing that number is definitely part of the problem. Let's eliminate the number parsing completely by using variables:

```
5 P=0
10 TM=TIMER:FORA=1TO1000
20 PRINT@P,"*";
30 NEXT:PRINT TIMER-TM
```

This gives us **229**, so it's a bit faster than the original. Now let's try the POKE version:

```
5 P=1024:
10 TM=TIMER:FORA=1TO1000
20 POKEP,106
30 NEXT:PRINT TIMER-TM
```

This gives us **400**, so it's faster than the original 655, but still nearly twice as slow as using PRINT@. But wait, there's still that 106 value. Let's replace that with a variable, too.

```
5 P=1024:V=106
10 TM=TIMER:FORA=1TO1000
20 POKEP,V
30 NEXT:PRINT TIMER-TM
```

This slows it down from 229 to **234**! We are now almost as fast as PRINT@! But now the POKE version has to look up two variables, while the PRINT@ version only looks up one, so that might give PRINT@ an advantage. Let's test this by making the PRINT@ version also use a variable for the character:

```
5 P=0:V$="*"
10 TM=TIMER:FORA=1TO1000
20 PRINT@P,V$;
30 NEXT:PRINT TIMER-TM
```

That slows it down to **231**. This seems to indicate the speed difference is really not between PRINT@ and POKE, but between how much number conversion of variable lookup each needs to do. You can use PRINT@ without having to look up the string to print ("*"), but POKE always has to either convert a numeric value (106) or do a variable lookup (L).

So why bother with POKE if the only advantage, so far, is that you can POKE to the bottom right character on the screen?

Because of PEEK.

PEEK lets us see what byte is at a specified memory location. If I were writing a game and wanted to tell if the player's character ran in to an enemy, I'd have to compare the player position (X/Y address, or PRINT@ location) with the locations of all the other objects. The more objects you have, the more compares you have to do and the slower your program becomes.

For example, here's a simple game where the player ("*") has to avoid four different enemies ("X"):

```
0 REM GAME.BAS
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=256+16
15 E1=32:E2=63:E3=448:E4=479
20
PRINT@P,"*";:PRINT@E1,"X";:PRINT@E2,"X";:PRINT@E3,"X";:PRINT
@E4,"X";
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$):IF LN=0 THEN 30
45 PRINT@P," ";
50 ONLN GOSUB100,200,300,400
60 IF P=E1 OR P=E2 OR P=E3 OR P=E4 THEN 90
80 GOTO 20
90 PRINT@267,"GAME OVER!":END
100 IF P>31 THEN P=P-32
110 RETURN
200 IF P<479 THEN P=P+32:RETURN
210 RETURN
300 IF P>0 THEN P=P-1
310 RETURN
400 IF P<510 THEN P=P+1
410 RETURN
```

In this example, the four enemies ("X") remain static in the corners, but if you move your player ("*") into one, the game will end.

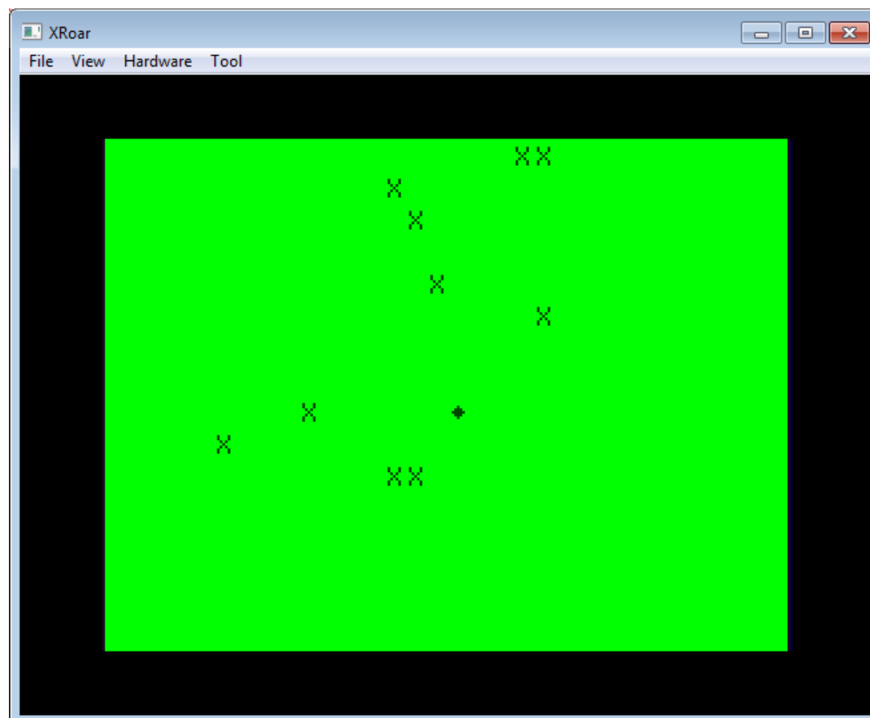
It's not much of a game, but with a few more lines you could make the enemies move around randomly or chase the player.

Take a look at line 60. Every move we have to compare the position of the player with four different enemies. This is a rather brute-force check. We could also use an array for the enemies. Not only would this simplify our code, but it would make the number of enemies dynamic.

Here is a version that lets you have as many enemies as you want. Just set the value of EN in line number 1 to the number of enemies-1.

Side Note: Arrays are base-0, so if you `DIM A(10)` you get 11 elements — A(0) through A(10). Thus, if you want ten elements in an array, you would do `DIM A(9)`, and cycle through them using base-0 like `FOR I=0 TO 9:PRINT A(I):NEXT I`.

```
0 REM GAME2.BAS
1 EN=10-1 'ENEMIES
2 DIM E(EN)
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=256+16
15 FOR A=0 TO EN:E(A)=RND(510):NEXT
20 PRINT@P,"*";
25 FOR A=0 TO EN:PRINT@E(A),"X";:NEXT
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$):IF LN=0 THEN 30
45 PRINT@P," ";
50 ONLN GOSUB100,200,300,400
60 FOR A=0 TO EN:IF P=E(A) THEN 90 ELSE NEXT
80 GOTO 20
90 PRINT@267,"GAME OVER!":END
100 IF P>31 THEN P=P-32
110 RETURN
200 IF P<479 THEN P=P+32:RETURN
210 RETURN
300 IF P>0 THEN P=P-1
310 RETURN
400 IF P<510 THEN P=P+1
410 RETURN
```



In 1980, this was a game.

Now the program is more flexible, but it has gotten slower. After every move, the code must now compare the locations of every enemy. This limits BASIC from being able to do a fast game with a ton of objects.

Which brings me back to PEEK… Instead of comparing the player against every enemy, all we really need to know is if the location of the player is where an enemy is. If we are using POKE to put the player on the screen, we know the location the player is, and can just PEEK that location to see if anything is there.

Let's change the program to use POKE and PEEK:

```
0 REM GAME2.BAS
1 EN=10-1 'ENEMIES
2 DIM E(EN)
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=1024+256+16:V=106:VS=96:VE=88
15 FOR A=0 TO EN:E(A)=1024+RND(511):NEXT
20 POKEP,V
25 FOR A=0 TO EN:POKEE(A),VE:NEXT
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$):IF LN=0 THEN 30
45 POKEP,VS
50 ONLN GOSUB100,200,300,400
60 IF PEEK(P)=VE THEN 90
80 GOTO 20
90 PRINT@267,"GAME OVER!":END
100 IF P>1024+31 THEN P=P-32
110 RETURN
200 IF P<1024+479 THEN P=P+32:RETURN
210 RETURN
300 IF P>1024+0 THEN P=P-1
310 RETURN
400 IF P<1024+510 THEN P=P+1
410 RETURN
```

Now, instead of looping through an array containing the locations of all the enemies, we simply PEEK to our new player location and if the byte there is our enemy value (VE, character 88), game over.

It should be a bit faster now.

We should also change the "1024+XX" things to the actual values to avoid doing math each time, but I was being lazy.

Now we know a way to improve the speed of reading key presses, and ways to use POKE/PEEK to avoid having to do manual comparisons of object locations. Maybe this will come in handy someday when you need to write a game where an asterisk is being chased by a bunch of Xs.

Until next time…

Comment 1

I think INSTR$ is fairly the best solution, but coming from dialects which do not have such a beast I tend to use an array mapping the INKEY$ character code to a value (for ON…GOTO or mapping back with another arrays to calculate the next state in position, compare to a limit, …). Sometimes a good trade-off in spite the waste of array space (if you can afford this). It would be interesting how competitive such a method is …

Part 4

INSTR and GOTO/GOSUB

Here's a quickie that discusses making `INSTR` faster, and `GOTO` versus `GOSUB`.

Side Note: In the code examples, I am using spaces for readability. Spaces slow things down, so instead of "`FOR A=1 TO 1000`" you would write "`FORA=1TO1000`". If you remove the unnecessary spaces from these examples, they get faster.

In the previous installment, I discussed ways to speed up doing things based on `INPUT` by using the `INSTR` command. `INSTR` will return the position of where a string is inside another string:

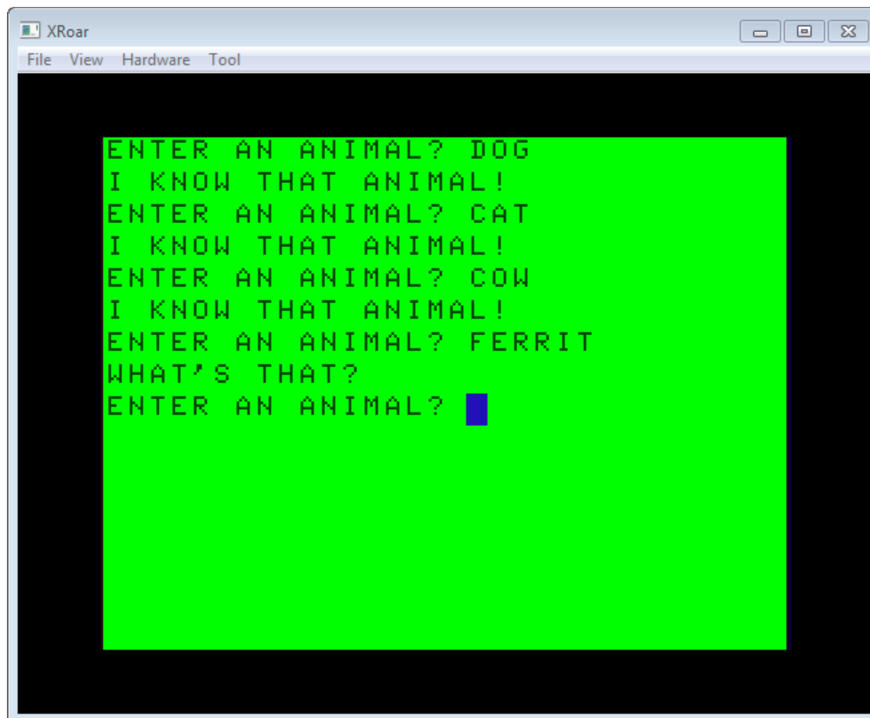
```
PRINT INSTR("ABC", "B") 2
```

Above, `INSTR` returns 2, indicating the string "B" was found starting at position 2 in the search string "ABC". If the string is not found, it returns zero:

```
PRINT INSTR("ABC", "X") 0
```

You could use this in weird ways. For instance, if you wanted to match only certain animals, you could do something like this:

```
10 INPUT "ENTER AN ANIMAL";A$
20 A=INSTR("CATDOGCOWCHICKEN", A$)
30 IF A>0 THEN PRINT "I KNOW THAT ANIMAL!" ELSE PRINT
"WHAT'S THAT?"
40 GOTO 10
```



INSTR can help you identify animals!

…but why would you want to do that? And, it just matches strings, so any combination that appears in the search string will be matched:

…or not.

Above, searching for "A" was a match, since there is an "A" in that weird animal string, as well as a "C". There was no "B", so…

Okay, never mind. Forget I mentioned it.

I am sure there are many good uses for INSTR, but I mostly use it to match single-letter commands (as mentioned previously) like this:

```
10 A$=INKEY$:IF A$="" THEN 10
20 LN=INSTR("ABCD", A$):IF LN=0 THEN 10
30 ON LN GOTO 100,200,300,400
```

Since INSTR returns 0 when there is no match, it's an easy way to validate that the character entered is valid.

According to the documentation in the CoCo 3 BASIC manual, the full syntax is this:

INSTR(start-position, search-string, target-string)

You can use the optional start position to begin scanning later in the string. For instance:

```
PRINT INSTR(3, "ABCDEF", "A")
```

That would print 0 since we are searching for "A" in the string "ABCDEF" starting at the third character (so, searching "CDEF").

The manual also notes conditions where a 0 can be returned:

- The start position is greater than the number of characters in the search-string: `INSTR(4, "ABC", "A")`
- The search-string is null: `INSTR("", "A")`
- It cannot find the target: `INSTR("ABC", "Z")`

I was surprised today to (re)discover that INSTR considers a null (empty) string to be a match, sorta:

```
PRINT INSTR("ABC", "") 1
```

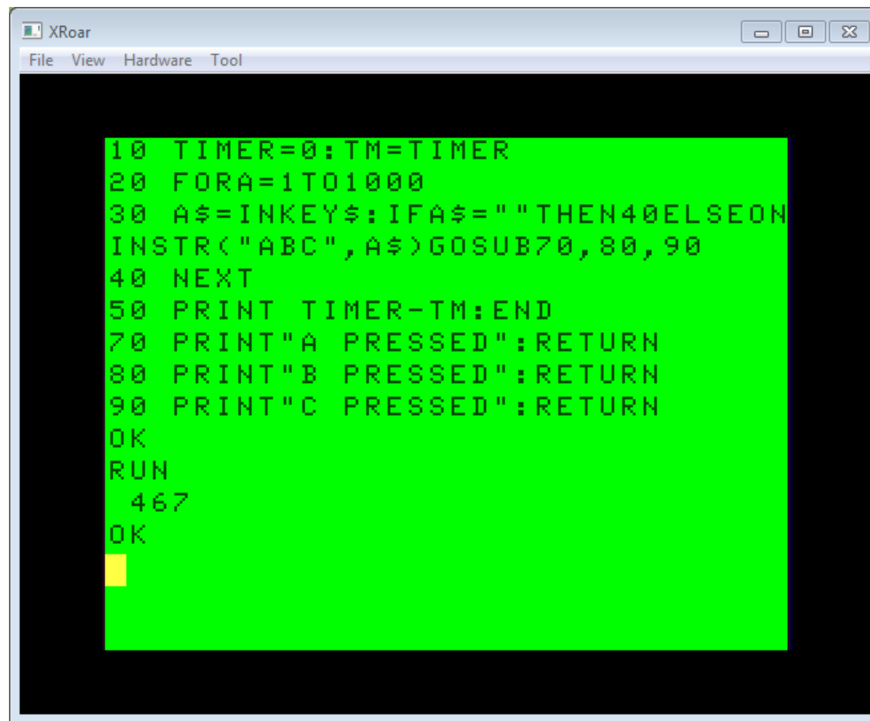
If the search string is empty, it returns with the current search position (which starts at 1, for the first character). This seems like a bug to me, but indeed, this behavior is the same in later, more advanced Microsoft BASICs.

I bring this up now because I was almost going to show you something really clever. Normally, I use INSTR with a string I get back from INKEY$. But, you can also use INKEY$ directly. And, since ON GOTO/GOSUB won't go anywhere if the value is 0, I thought it might be clever to use it like this:

```
10 ON INSTR("ABCD",INKEY$) GOTO 100,200,300,400
```

…and this is smaller and much faster and works great … if there is a key waiting! If no key is waiting, INKEY$ returns a null ("") and … INSTR returns a 1, and then ON GOTO goes to 100 even though that was not the intent.

Darnit. I thought I had a great way to speed things up. Consider the speed of this version.



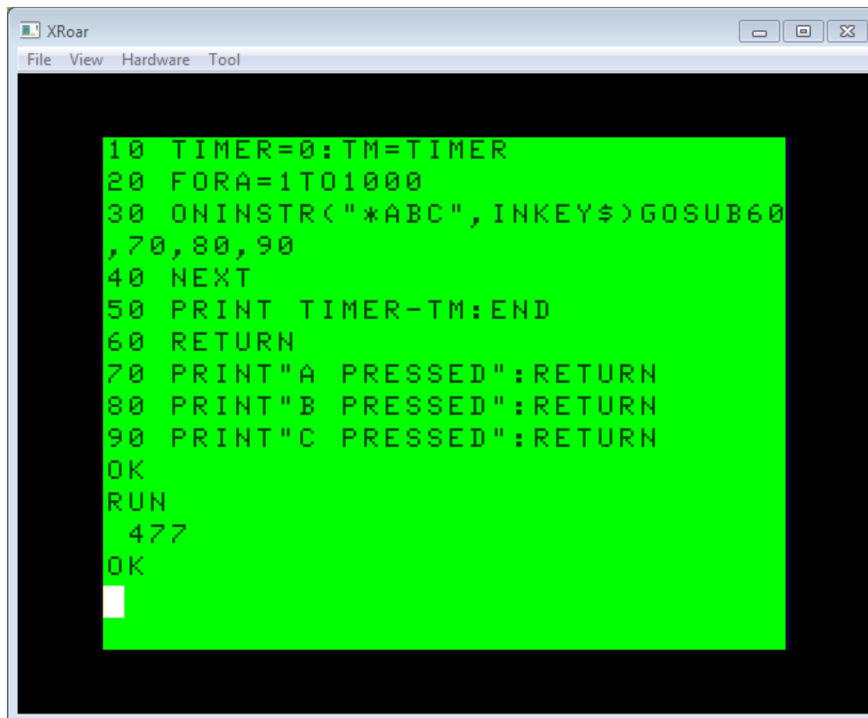
INSTR example … workaround for not using a variable.

I thought by replacing line 30 with…

```
ON INSTR("ABC",INKEY$)GOSUB70,80,90
```

…I would be set. But, since an empty `INKEY$` is returning "", it's always `GOSUB`ing to line 70.

I tried a hacky workaround, by adding a bogus character to the start of the string, and making that `GOSUB` to a `RETURN` located real close to that code (so it didn't have to search as far to find it):



INSTR example.

…but, the overhead of that extra `GOSUB`/`RETURN` that happens EVERY TIME there is no key waiting was enough to make it slightly slower. If it wasn't for that, we could do this maybe 30% faster and use less variables.

So, unfortunately, I guess I have no optimization to show you… Just a failed attempt at one.

But wait, there's more!

I posted about this on the CoCo mailing list and in the CoCo Facebook group to figure out if this behavior was a bug. There were several responses confirming this behavior in other versions of BASIC and languages.

On the list, Robert Hermanek responded with a great solution:

> *Issue is just getting a return of 1 when searching for empty string? Then why not:*

```
10 ON INSTR(" ABC",INKEY$) GOTO 10,100,200,300...
```

> *notice the space before A.*

His brilliant suggestion works by adding a bogus character to the search string, and making any match of that string (or "") `GOTO` the same line. Thus, problem solved!

This won't work with `ON GOSUB` since every `GOSUB` expects a `RETURN`. Each time you use `GOSUB`, it takes up seven bytes (?) of memory to remember where to `RETURN` to in the program. If you did something like this, you'll see the issue:

```
10 PRINT MEM:GOSUB 10
```

I make a quick change to my program to use GOTO instead:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 ON INSTR(" ABC",INKEY$) GOTO 40,70,80,90
40 NEXT
50 PRINT TIMER-TM:END
70 PRINT"A PRESSED":GOTO 40
80 PRINT"B PRESSED":GOTO 40
90 PRINT"C PRESSED":GOTO 40
```

For my timing test inside the FOR/NEXT loop, I made the first GOTO point to the NEXT in line 40, but if I wanted to wait "forever" until a valid key was pressed, I would make that 30.

This version shows a time of **412**, so let's compare that to doing it with A$:

```
10 TIMER=0:TM=TIMER
20 FORA=1 TO 1000
30 A$=INKEY$:IFA$="" THEN 40 ELSE ON INSTR("ABC",A$) GOTO
70,80,90
40 NEXT
50 PRINT TIMER-TM:END
70 PRINT"A PRESSED":GOTO 40
80 PRINT"B PRESSED":GOTO 40
90 PRINT"C PRESSED":GOTO 40
```

This produces **486**. We now have a way to avoid using A$ and speed up code just a bit.

This made me wonder … what is faster? Using GOTO, or doing a GOSUB/RETURN? Let's try to predict…

Both GOTO and GOSUB will have to take time to scan through the program to find the destination line, but GOSUB will also have to take time to store the "where are we" return location so RETURN can get back there. This makes me think GOSUB will be slower.

BUT, we need a GOTO to return from a GOTO, and a RETURN to return from a GOSUB. GOTO always has to scan through the program line by line to find the destination, while RETURN just jumps back to a location that was saved by GOSUB. So, if we have to scan through many lines, the return GOTO is probably slower than a RETURN.

Let's try.

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 GOSUB 1000
40 NEXT
50 PRINT TIMER-TM:END
1000 RETURN
```

That prints **140**.

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 GOTO 1000
40 NEXT
50 PRINT TIMER-TM:END
1000 GOTO 40
```

That prints **150**.

I expect it is because line 1000 says "GOTO 40" and since 40 is lower than 1000, BASIC has to start at the top and go line by line looking for 40. If you GOTO to a higher number, it starts from the current line and moves forward. The speed of GOTO (and GOSUB) varies based on where the lines are:

GOTO should be quick when going to a line right after it:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 GOTO 31
31 GOTO 32
32 GOTO 33
40 NEXT
50 PRINT TIMER-TM:END
```

That prints **170**.

Line 30 has to scan one line down to find 31, then 31 scans one line down to find 32, and 32 scans one line down to find 40.

But if you change the order of the GOTOs:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 GOTO 32
31 GOTO 40
32 GOTO 31
40 NEXT
50 PRINT TIMER-TM:END
```

…that prints **175**.

It is the same number of GOTOs, but line 30 scans two lines ahead to find 32, then 32 has to start at the top and scan four lines in to find line 31, then line 31 has to scan two lines ahead to find 40.

If we add more lines (even REMs), more things have to be scanned:

```
0 REM
1 REM
2 REM
3 REM
4 REM
5 REM
6 REM
7 REM
8 REM
9 REM
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 GOTO 32
31 GOTO 40
32 GOTO 31
40 NEXT
50 PRINT TIMER-TM:END
```

We are now up to **192** and the one with the lines in order is still **170**.

The more lines `GOTO` (or `GOSUB`) has to search, the slower it gets. So while MAYBE there might be a case where a `GOTO` could be quicker than a `RETURN`, it seems that even with a tiny program, `GOSUB` wins.

So … the question is, is the time we save doing the `INKEY` like this:

```
30 ON INSTR(" ABC",INKEY$) GOTO 40,70,80,90
```

…going to offset the time we lose because those functions all have to `GOTO` back, rather than using a `RETURN`?

If this was a normal "wait for a keypress" then it probably wouldn't matter much. We are just waiting, so there is no time to save by making that faster.

If we were reading keys for an action game, the actual "is there a keypress?" code would be faster, giving more time for the actual program. But, every time a key was pressed, the time taken to get in and out of that code would be slower. I guess it depends on how often the key is pressed.

A game like Frogger, where a key would be pressed every time the frog jumps to the next spot, might be worse than a game like Pac-Man where you press a direction and then don't press anything again until the character is at the next intersection to turn down.

I am not sure how I would benchmark that, yet, but let's try this… We'll modify the code so "" actually is honored as an action:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 ON INSTR(" UDLR",INKEY$) GOTO 100,200,300,400,500
40 NEXT
50 PRINT TIMER-TM:END
100 REM IDLE LOOP
110 GOTO 40
200 REM MOVE UP
210 GOTO 40
300 REM MOVE DOWN
310 GOTO 40
400 REM MOVE LEFT
410 GOTO 40
500 REM MOVE RIGHT
510 GOTO 40
```

Now if no key is waiting (“”), `INSTR` will return a 1 causing the code to `GOTO 100` where the background (keep objects moving, animate stars, etc.) action would happen. Any other value would go to the handler for up, down, left or right.

This prints **457**. Doing the same thing with `GOSUB`:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 ON INSTR(" UDLR",INKEY$) GOSUB 100,200,300,400,500
40 NEXT
50 PRINT TIMER-TM:END
100 REM IDLE LOOP
110 RETURN
200 REM MOVE UP
210 RETURN
300 REM MOVE DOWN
310 RETURN
400 REM MOVE LEFT
410 RETURN
500 REM MOVE RIGHT
510 RETURN
```

…prints **487**. It appears `GOSUB`/`RETURN` is slower for us than `GOTO` here. But why? `GOSUB` seemed faster in the first example. Is `ON GOSUB` slower than `ON GOTO`?

Quick side test:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 ON 1 GOSUB 1000
40 NEXT
50 PRINT TIMER-TM:END
1000 RETURN
```

That prints **249**. `GOSUB` by itself was 140, so `ON GOSUB` is much slower.

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 ON 1 GOTO 1000
40 NEXT
50 PRINT TIMER-TM:END
1000 GOTO 40
```

…also prints **249**, and GOTO by itself was 150. I am a bit surprised by this. I will have to look in to this further for an explanation.

But I digress…

We can still slow down GOTO. If we had a bunch of extra lines for GOTO to have to scan through:

```
0 REM
1 REM
2 REM
3 REM
4 REM
5 REM
6 REM
7 REM
8 REM
9 REM
```

…that will slow down every GOTO to an earlier number. With those REMs added, we have:

- GOTO/GOTO version with REMs: **473** (up from 457 without REMs)
- GOSUB/RETURN version with REMs: **487** (it never passes through the REMs)

It appears that, while GOSUB/RETURN may be faster on it's own, when I put it in this test program, GOTO/GOTO is slightly faster, but that can change depending on how big the program is. More research is needed…

So I guess, for now, I'm going to avoid using a variable for INKEY$ and use GOTO/GOTO for my BASIC game…

Until next time…

Comment 1

I suspect the reason the ON…GOSUB and ON…GOTO variants are substantially slower is due to the expression evaluation that has to be run. That's a fairly slow process, especially given that in the case of "ON 1", there's an ASCII to floating point conversion. I suspect that the expression evaluation swamps both line number searches and the GOSUB/RETURN stack handling overhead which probably explains why there's no apparent difference between ON…GOTO and ON…GOSUB in the specific case tested. I would expect larger programs should pan out in favor of ON…GOSUB, though, at least if the extra program code is before the INKEY loop.

I did some experiments with similar benchmarks and got the same result you did. See http://lost.l-w.ca/0x05/optimizing-color-basic-on-goto-vs-on-gosub/ for my analysis. Basically, I also got no measurable difference between ON…GOTO and ON…GOSUB.

I think your ON…GOSUB variant that tested as slower than the ON…GOTO one has to do with the GOSUB/RETURN

happening for every idle loop instead of just bailing directly to the “NEXT” instruction in the idle case. After all, the idle case is what is going to execute more often than not. In a non-trivial program, that difference probably disappears in the noise but that in itself may be a good reason to use the GOTO variant. Key presses will be the exception rather than the norm so optimizing for the idle case probably gives you a better speedup overall. Of course, one should benchmark the real program to see which variant is better in a given circumstance.

Comment 2

Awesome, William! I am reading your write-up right now. I will post questions over there.

Comment 3

For Optimal speed try this:

```
0 GOTO10
1 NEXT:GOTO50
10 TIMER=0:TM=TIMER
20 FORA=1TO1000
30 ON1GOTO1
40 NEXT
50 PRINT TIMER-TM:END
```

Comment 4

Hey James! I was hoping you were out here somewhere. I’ve been meaning to e-mail you some questions.

Doesn’t that just bypass the NEXT in 40? And the idea is that GOTO to the second line of the program is about as fast as you can get unless it’s the first line or the next line after GOTO.

I have an updated benchmark test that averages out multiple runs, and when I get caught up (posting one a day until I am), it would be neat to see what else you would do with some of the things presented.

Comment 5

> Doesn’t that just bypass the NEXT in 40?

Just in this special case line 40 is never reached. Generally NEXT may appear multiple times matching the last FOR in run-time … needed in situations where ON..GOTO might fall through.

Comment 6

My suggestion has a number of techniques which I believe might help with speed, First put all common subroutines at the top of the program (to shorten the search interpreted Basic must do to find a line number). Second, if you going to GOTO or ON/GOTO to subroutines from within a tight FOR/NEXT loop, then why not use additional NEXT commands to return you to that loop. This way you save the interpreter having to deal with an entire additional GOTO command and line number for each visit to the subroutine (at least until the last element of the FOR/NEXT loop). This is possible because basic doesn’t care how many NEXT commands you put in a program for any FOR loop, so why not use NEXTs to perform the double functions of returning you from subroutines and also iterating the loop, if you can structure a program that way.

Comment 7

Could you show me an example of using NEXT to get back from the ON/GOTO from within a loop? Also, I gather that, since GOTO/GOSUB stops scanning forward as soon as it finds a line number greater than what you ask for, it makes sense to put all the “run once” at the end of the program.

Comment 8

I think something like that was in mind:

```
20 FOR A=. TO 1: A=. : REM FOREVER
30 ON INSTR(" UDLR",INKEY$) GOTO 100,200,300,400,500
35 REM FALL THROUGH ACTION
40 NEXT
50 END
100 REM IDLE LOOP
110 NEXT
200 REM MOVE UP
210 NEXT
300 REM MOVE DOWN
310 NEXT
400 REM MOVE LEFT
410 NEXT
500 REM MOVE RIGHT
510 NEXT
```

Compared to the `ON GOSUB` this is some kind of "redo" or "loop retry" not reaching the fall through action in 35.

Comment 9

How about using a single `GOSUB` to the inkey and then return from each option:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 GOSUB 90
40 NEXT
50 PRINT TIMER-TM:END
90 REM ALTERNATIVE BLENDED STRUCTURE
95 ON INSTR(" UDLR",INKEY$) GOTO 100,200,300,400,500
100 REM IDLE LOOP
110 RETURN
200 REM MOVE UP
210 RETURN
300 REM MOVE DOWN
310 RETURN
400 REM MOVE LEFT
410 RETURN
500 REM MOVE RIGHT
510 RETURN
```

Comment 10

"Just when I thought I was out…they pull me back in." I am intrigued. Forward searching has to happen for GOTO or GOSUB, so that part shouldn't be too different. But the return pops back, preventing the need to start at the top and search forward each time. I will try to find some time to experiment with this.

Part 5

HEX versus DECimal Numbers

As Barbie once said*…

> Math is hard!

While Mattel's Math-Is-Hard Barbie never quite made the splash the marketing team had hoped for, her sentiment lives on.

Side Note: This is in reference to a the Teen Talk Barbie doll released in 1992, and out of the 270 phrases the doll could say, that was not one of them. The real quote was "Math class is tough!"

Earlier in this series, I touched on the fact that dealing with numbers is time-consuming for BASIC. Something as simple as B=65535 takes time to process as the interpreter translates that base-10 decimal number into an internal floating point value. The more digits, the more work. For instance:

```
0 REM NUMBERS.BAS
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 B=1
40 NEXT
50 PRINT TIMER-TM
```

That prints a value of **183**. If you change line 3 to read "B=12345" the number jumps to **485**. You can see the increase:

```
B=1-183
B=12-262
B=123-337
B=1234-408
B=12345-485
```

Obviously, the more numbers to parse and convert, the more time it will take. It also seems to matter if the value has a decimal point in it:

```
B=1.0-403
B=1.1-476
```

Even though that is only three characters to process, it takes longer than B=123. Clearly, more work is being done on floating point values. Even though all Color BASIC numbers are represented internally as floating point, it still makes sense to avoid using them unless you really need them.

You can also represent a base-16 number in hexadecimal. For the value of 1, it feels like parsing "&H1" should take longer than parsing "1". Let's try:

```
B=&H1-180
B=&H12-175
B=&H123-200
B=&H1234-203
```

It seems that parsing a hexadecimal value is much faster than dealing with base-10 values. Using this, you could speed up a program just by switching to hex, provided that your numbers are between 0 and 65535 (the values that can be represented in hex). I was surprised to see that negative values also work:

```
B=&HFFFF-201
B=-&HFFFF-230
```

It seems dealing with the negative takes a bit of more time, though, so it makes sense to avoid using them unless you really need them.

With this in mind, let's test a `FOR`/`NEXT` loop:

```
10 TIMER=0:TM=TIMER
20 FOR A=&H1 TO &H3E8
30 B=&H1
40 NEXT
50 PRINT TIMER-TM
```

This prints **182**, which is basically the same speed as the original that used `0 TO 1000`. I guess hexadecimals don't really help out `FOR`/`NEXT`.

Why? Because the `FOR`/`NEXT` statement is only parsed once, then the loop counters are set up and done. It is probably a tad faster to use hex, but that savings only happens once in the "do it 1000 times" test.

But, as you see, USING the variables gets faster. Any place we use a number, it seems using a hex version of that number may speed it up:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 IF A>&HFF THEN REM
40 NEXT
50 PRINT TIMER-TM
```

This prints **278**. Doing it with A>255 prints **427!** Imagine if you could speed up every time you used a number in your code:

```
10 TIMER=0:TM=TIMER
20 FOR A=1 TO 1000
30 PRINT@&H20,"HELLO"
40 NEXT
50 PRINT TIMER-TM
```

That prints **391**, but changing it to `PRINT@32` prints **469**! If you use a bunch of `PRINT@`s in your code, you can speed them up just by switching to hex!

Math could be accelerated, too, simply due to the number conversion being faster. The more digits, the better advantage hex has:

```
B=A+&H270F-285
B=A+9999-483
```

And the more numbers, the more time you can save by using hex. A common `PRINT` thing is to use the length of a string to figure out how to center it on the screen:

```
0 REM NUMBERS.BAS
10 TIMER=0:TM=TIMER
15 CLS:A$="HELLO, WORLD!":LN=LEN(A$)
20 FOR A=1 TO 1000
25 PRINT@32*8+16-LN/2,A$
40 NEXT
50 PRINT TIMER-TM
```

That prints **1284**. Converting line 24 to HEX:

```
25 PRINT@&H20*&H8+&H1-LN/&H2,A$
```

And now it prints **1097**.

In a game where you might be PRINTing things on the screen constantly, those savings could really add up.

Pity that math is hard, else we could just use hex in our programs and get a free speed boost.

Until next time…

Comment 1

One place where a decimal point will give an improvement is when using the value 0 in an expression. Basic will accept a stand-alone decimal point as the number 0, but it will process it faster than the '0' character.

Try comparing the speed of:

```
IF N < 0 THEN ...
```

with that of:

```
IF N < . THEN ...
```

Comment 2

Mind blown.

Comment 3

You want something to really blow your mind? You can put spaces in the middle of numbers. You can also put spaces in the middle of variable names.

Comment 4

There is absolutely no way that is true. ;)

Comment 5

```
A B = 10
```

```
PRINT A B
10
A = 1 2 3 4 5
PRINT A
12345
```

That's awesome. Why the heck does that work?

Comment 6

The CHRGET subroutine for Microsoft-based interpreters parsing the program text skips every space character. The one that are needed to prevent the tokenizer to misinterpret keywords or variable names aren't necessary at all after a line is tokenized. These blanks are usually kept for ease of editing. Spaces in string constants are directly handled by the expression evaluation without using CHRGET. It's just a side-effect due to the lack of a strong lexical analysis.

Comment 7

Thanks, Johann! I don't know your name. How did you come to know the internals of the interpreter?

Comment 8

Did a lot on Commodore-based systems, digging into the interpreter to merge Basic and machine code stuff, to accomplish parameter passing and return values, extending the interpreter, improved string garbage collection and so on. Later I got a Dragon32 which fascinates my (my preferred CPU) I stumbled into the Extended Color BASIC and saw all the similarities … I could do the same interfacing, nearly the same data structures, just other addresses (and of course the endianess …).

Comment 9

I started on a VIC-20, and preferred the Extended BASIC on the CoCo. It wasn't until recent years that I found out Commodore BASIC was Microsoft. Any idea why it used GET$ instead of INKEY$?

Comment 10

The GET command is typical for the 6502 branch of the early MS Basic. I think GET was simply part of the unified I/O commands (based on logical file numbers on top of static device numbers) which allowed to read a single character from the standard input device (the keyboard). Output redirection with CMD can easily achieved. As opposed INKEY$ is a string function, not a command which has to be invented to the read the keyboard because the above mentioned file number layer is missing. The basic I/O command are reduced to PRINT and INPUT. Single byte input was not a necessary. But Commodore computers (mainly business oriented) with their IEEE interface needed the single-byte-read-ability to communicate with devices in a very distinct and controlled manner (later on at home computer times this device was only the floppy which demanded this kind of operation).

I hope this meets the point.

Comment 11

So in my VIC days, GET A$ was kind of an implied GET #stdin,A$ read?

Comment 12

> So in my VIC days, GET A$ was kind of an implied GET #stdin,A$ read?

Exactly, if you open the keyboard device (number 0) you can do this

```
10 OPEN 1,0
```

```
20 GET#1,A$:IF A$="" GOTO20
30 PRINT A$
```

Alas, there no concept of STDIN as opposed to STDOUT which is controlled by the command CMD. So GET A$ (without hash argument) is always bound to the keyboard. But this is just a limitation of Basic itself, the underlying Kernel on Commodores actually do keep a "current input device".

Comment 13

I understand. I only had the Datasette and a cheesy thermal printer in my VIC days, so I never learned much about I/O. It all makes much more sense to me today than it did back then :)

Comment 14

How did you figure this out???

Comment 15

Thanks for commenting! I am learning much.

Comment 16

Son of a gun. A quick test in my benchmark program using "Z=0" showed 178, and "Z=." showed 141.

It looks like you can do PRINT@.,"HELLO" too. Wild.

Comment 17

I read about it someplace, but don't remember where. It was probably part of a discussion on the CoCo mailing list.

Comment 18

The reason -&HFFFF is slower is actually pretty straight forward yet also counter-intuitive. It actually evaluates the minus as unary negation and the &HFFFF is converted independently. Now negation is practically instant in the floating point representation used. However, it does require an extra trip through the expression evaluator.

Also, as a side note, on the Coco3, &H (and &O) can be used for 24 bit values. They expanded it so that LPOKE and LPEEK could be used with hex addresses across the whole address space.

Part 6

Size Matters. Or Space Matters. You decide.

Sometimes we want to optimize for code space, and other times for variable and string space. For example, if you want to create a 32 character string like this:

```
<>
```

…you could either declare it as a static string:

```
A$ = "<    >"
```

Or build it programatically like this:

```
A$="<"+STRING$(30,"-")+">"
```

The second version takes about 16 bytes less of program space because the string is generated dynamically in string memory rather than being stored in the tokenized BASIC program.

Doing it the second way seems like a good idea, but keep in mind when you make this string, somewhere in string memory will be those 32 characters, PLUS you still have the BASIC statements that created it. It's actually larger, overall, to do it this way.

BUT, any temporary strings like that might make sense to create on-the-fly as you need them since that memory can be reused by other strings.

```
10 A$="<
20 PRINT A$:PRINT "MAIN MENU":PRINT A$
30 INPUT "COMMAND",-C$
```

In the above example, A$ points to that sequence of characters INSIDE the BASIC program itself. It is always there. But, if you generated the string only when needed, the memory used by A$ could be used for other purposes:

```
10 A$="<"+STRING$(30,"-")+">"
20 PRINT A$:PRINT "MAIN MENU":PRINT A$
30 INPUT "COMMAND",-A$
```

Above, A$ is allocated and turned into the long 32-character string, printed, and then the memory used by A$ can be reused by `INPUT`. I suppose just setting it to A$="" might give it back, too.

This would come with a speed penalty since the creation and destruction of strings takes more CPU time than just using a static string.

I think I may have also mentioned that, even if a string is part of the BASIC program, if you do anything to it, it has to duplicate it in string memory which creates a second copy of it:

```
10 A$="<
20 A$=A$+"HELLO"
```

Above, A$ initially starts out pointing inside the program itself, taking up none of that string memory. At line 20, the entire A$ gets copied in to string memory and then the extra characters are added to it. At that point, that string is now using over twice the memory (program space plus string space).

Let's try to prove that. The `CLEAR` command is used to reserve memory for strings. By default, 200 bytes are reserved. We can change that by doing `CLEAR 0`. Here is a program that has no string memory, yet it works because the string is inside the program space:

```
10 CLEAR 0
20 A$="<--
```

If you run this, you can `PRINT A$` and prove it exists, but the moment you try to declare a second string like `B$="HELLO"` or even manipulate A$ like `A$=A$+""` you will get an Out of String Space errors (?OS ERROR):

Proving strings can live inside program space.

Sometimes you choose speed over size, and sometimes you choose size over speed. Thus, you can optimize for speed (which we have been doing so far), or optimize for size.

But I digress.

Elementary, my dear DATA.

Today I want to discuss `DATA` statements. In my assembly language series, I showed how the lwasm assembler can generate a small BASIC program that has the assembly code in `DATA` statements, and a small

loader which will READ them and POKE them in to memory:

```
10 READ A,B
20 IF A=-1 THEN 70
30 FOR C = A TO B
40 READ D:POKE C,D 50 NEXT C
60 GOTO 10
70 END
80 DATA
16128,16167,142,63,14,166,128,39,6,173,159,160,2,32,246,57,8
4,104,105,115,32,105,115,32,97,32,115,101,99,114,101,116,32,
109,101,115,115,97,103,101,46,0,-1,-1
```

DATA statements can contain base-10 numbers, base-16 hexadecimal numbers, or strings (and I guess base-8 octal numbers too, but who would do that?). This means you could have the data stored as numbers:

```
   100 DATA 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
   100 DATA
&H0,&H2,&H3,&H4,&H5,&H6,&H7,&H8,&H9,&HA,&HB,&HC,&HD,&H
E,&HF
```

…or as strings like "FE" or "1F" that you could READ and convert to hex numbers in the loader:

```
   100 DATA
59,4F,55,20,4D,55,53,54,20,42,45,20,42,4F,52,45,44
```

When it comes to a size, hexadecimal numbers without the "&H" in front are always smaller than their base-10 equivalent. Single-digit decimal values 0-9 are single-digit 0-9 in hex. Double-digit decimal values 10-15 are represented by single-digit hexadecimal values A-F. Every time a value from 10-15 appears, representing it in decimal takes up twice as much space. And for three-digit decimal values 100-255, those are two-digit hex values 64-FF.

If you store the data as strings, like this:

```
   100 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
   101 DATA
10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F
```

…you can read each string in, and convert it to a number by adding "&H" to the start and using the VAL() function:

```
   READ B$:B=VAL("&H"+B$)
```

For the decimal and hexadecimal versions, you just read it as a number:

```
   READ B
```

The smallest version would be the string approach, since three-digit numbers can be represented with two digits. But, doing the string conversion with VAL() makes it slower.

The fastest version would be using hexadecimal numbers since BASIC can parse hex values faster than base-10 numbers. But, this is the largest version since 255 in decimal (3 characters) or FF as a hex string (2 characters) would be represented as &HFF as a hex number (4 characters). Those numbers would take up twice as much space as the string version!

In the middle is base-10 numbers. It's not the largest, or the smallest, or the fastest or the slowest. It makes an ideal compromise.

Let's do a test. I have `DATA` statements representing values from 0 to 255. I have three versions: the first will use base-10 numbers, the second will use hexadecimal numbers, and the third will use strings that are just the hex part of the "&H" number.

Base 10 Numbers

```
0 REM DATADEC.BAS
10 TIMER=0:TM=TIMER
20 FOR A=0 TO 255
30 READ B
40 NEXT
50 PRINT TIMER-TM
60 END
100 DATA 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
101 DATA 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
102 DATA 32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47
103 DATA 48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63
104 DATA 64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79
105 DATA 80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95
106 DATA
96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111
107 DATA
112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,
127
108 DATA
128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,
143
109 DATA
144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,
159
110 DATA
160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,
175
111 DATA
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,
191
112 DATA
192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,
207
113 DATA
208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,
223
114 DATA
224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,
239
115 DATA
240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,
255
```

Hexadecimal Base-16 Numbers

```
0 REM DATAHEX.BAS
10 TIMER=0:TM=TIMER
20 FOR A=0 TO 255
30 READ B
40 NEXT
50 PRINT TIMER-TM
60 END
100 DATA
&H0,&H1,&H2,&H3,&H4,&H5,&H6,&H7,&H8,&H9,&HA,&HB,&HC,&HD,&HE,
&HF
101 DATA
&H10,&H11,&H12,&H13,&H14,&H15,&H16,&H17,&H18,&H19,&H1A,&H1B,
&H1C,&H1D,&H1E,&H1F
102 DATA
&H20,&H21,&H22,&H23,&H24,&H25,&H26,&H27,&H28,&H29,&H2A,&H2B,
&H2C,&H2D,&H2E,&H2F
103 DATA
&H30,&H31,&H32,&H33,&H34,&H35,&H36,&H37,&H38,&H39,&H3A,&H3B,
&H3C,&H3D,&H3E,&H3F
104 DATA
&H40,&H41,&H42,&H43,&H44,&H45,&H46,&H47,&H48,&H49,&H4A,&H4B,
&H4C,&H4D,&H4E,&H4F
105 DATA
&H50,&H51,&H52,&H53,&H54,&H55,&H56,&H57,&H58,&H59,&H5A,&H5B,
&H5C,&H5D,&H5E,&H5F
106 DATA
&H60,&H61,&H62,&H63,&H64,&H65,&H66,&H67,&H68,&H69,&H6A,&H6B,
&H6C,&H6D,&H6E,&H6F
107 DATA
&H70,&H71,&H72,&H73,&H74,&H75,&H76,&H77,&H78,&H79,&H7A,&H7B,
&H7C,&H7D,&H7E,&H7F
108 DATA
&H80,&H81,&H82,&H83,&H84,&H85,&H86,&H87,&H88,&H89,&H8A,&H8B,
&H8C,&H8D,&H8E,&H8F
109 DATA
&H90,&H91,&H92,&H93,&H94,&H95,&H96,&H97,&H98,&H99,&H9A,&H9B,
&H9C,&H9D,&H9E,&H9F
110 DATA
&HA0,&HA1,&HA2,&HA3,&HA4,&HA5,&HA6,&HA7,&HA8,&HA9,&HAA,&HAB,
&HAC,&HAD,&HAE,&HAF
111 DATA
&HB0,&HB1,&HB2,&HB3,&HB4,&HB5,&HB6,&HB7,&HB8,&HB9,&HBA,&HBB,
&HBC,&HBD,&HBE,&HBF
112 DATA
&HC0,&HC1,&HC2,&HC3,&HC4,&HC5,&HC6,&HC7,&HC8,&HC9,&HCA,&HCB,
&HCC,&HCD,&HCE,&HCF
113 DATA
&HD0,&HD1,&HD2,&HD3,&HD4,&HD5,&HD6,&HD7,&HD8,&HD9,&HDA,&HDB,
&HDC,&HDD,&HDE,&HDF
```

```
114 DATA
&HE0,&HE1,&HE2,&HE3,&HE4,&HE5,&HE6,&HE7,&HE8,&HE9,&HEA,&HEB,
&HEC,&HED,&HEE,&HEF
115 DATA
&HF0,&HF1,&HF2,&HF3,&HF4,&HF5,&HF6,&HF7,&HF8,&HF9,&HFA,&HFB,
&HFC,&HFD,&HFE,&HFF
```

String HEX Numbers

```
0 REM DATASTR.BAS
10 TIMER=0:TM=TIMER
20 FOR A=0 TO 255
30 READ B$:B=VAL("&H"+A$)
40 NEXT
50 PRINT TIMER-TM
60 END
100 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
101 DATA 10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F
102 DATA 20,21,22,23,24,25,26,27,28,29,2A,2B,2C,2D,2E,2F
103 DATA 30,31,32,33,34,35,36,37,38,39,3A,3B,3C,3D,3E,3F
104 DATA 40,41,42,43,44,45,46,47,48,49,4A,4B,4C,4D,4E,4F
105 DATA 50,51,52,53,54,55,56,57,58,59,5A,5B,5C,5D,5E,5F
106 DATA 60,61,62,63,64,65,66,67,68,69,6A,6B,6C,6D,6E,6F
107 DATA 70,71,72,73,74,75,76,77,78,79,7A,7B,7C,7D,7E,7F
108 DATA 80,81,82,83,84,85,86,87,88,89,8A,8B,8C,8D,8E,8F
109 DATA 90,91,92,93,94,95,96,97,98,99,9A,9B,9C,9D,9E,9F
110 DATA A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,AA,AB,AC,AD,AE,AF
111 DATA B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,BA,BB,BC,BD,BE,BF
112 DATA C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,CA,CB,CC,CD,CE,CF
113 DATA D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,DA,DB,DC,DD,DE,DF
114 DATA E0,E1,E2,E3,E4,E5,E6,E7,E8,E9,EA,EB,EC,ED,EE,EF
115 DATA F0,F1,F2,F3,F4,F5,F6,F7,F8,F9,FA,FB,FC,FD,FE,FF
```

If we look at the size JUST the `DATA` statement lines take up (lines 100-115), here is the size breakdown:

```
DATADEC.BAS - SPEED 78, SIZE 1010
DATAHEX.BAS - SPEED 49, SIZE 1360
DATASTR.BAS - SPEED 109, SIZE 848
```

As you can see, using hex values is over twice as fast as using string versions and converting them to hex.

For size, using strings is about 15% smaller in my test program than using decimal values.

If load time is important, use hex. If program space is important, use strings. Otherwise, normal decimal values are a good compromise between speed and size.

Bonus Data

One more thing… If we are going to use strings anyway, we could save more space by making the hex strings long, and parsing through them to pull out the individual hex values. Every number has to be two characters

(00, 01, 02 … 0E, 0F) and this additional string parsing makes it even slower, but if code size is most important, try this:

```
0 REM DATASTR2.BAS
10 TIMER=0:TM=TIMER
20 FOR A=0 TO 15
30 READ B$:FOR I=1 TO 32 STEP 2:B=VAL("&H"+MID$
(B$,I,2)):NEXT
40 NEXT
50 PRINT TIMER-TM
60 END
100 DATA 000102030405060708090A0B0C0D0E0F
101 DATA 101112131415161718191A1B1C1D1E1F
102 DATA 202122232425262728292A2B2C2D2E2F
103 DATA 303132333435363738393A3B3C3D3E3F
104 DATA 404142434445464748494A4B4C4D4E4F
105 DATA 505152535455565758595A5B5C5D5E5F
106 DATA 606162636465666768696A6B6C6D6E6F
107 DATA 707172737475767778797A7B7C7D7E7F
108 DATA 808182838485868788898A8B8C8D8E8F
109 DATA 909192939495969798999A9B9C9D9E9F
110 DATA A0A1A2A3A4A5A6A7A8A9AAABACADAEAF
111 DATA B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF
112 DATA C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF
113 DATA D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF
114 DATA E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF
115 DATA F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
```

DATASTR2.BAS – speed **172**, size 624

By removing all those commas, it's the smallest data size yet. And, since the longest line you can type in BASIC is 249 characters…

BASIC allows for typing up to 249 characters on a line.

…you could really back some data in to it.

Side Note: The BASIC editor allows for 249 characters, but when you press ENTER, the line is tokenized. Keywords like `PRINT` get reduced to smaller tokens. You may have typed a five-character keyword (taking up part of that 249 byte buffer), but when you press ENTER, that five characters may be converted to a one-byte token. This means it's possible for a BASIC line to contain more valid code than you could actually type. There have been utilities for BASIC (such as **Carl England**'s CRUNCH) that do this, packing program lines as big as they can be, and making them un-editable since the moment you try, they get detokenized and you lose anything past 249 characters. We'll have to discuss this in a later installment.

With that in mind, we could pack any type of `DATA` in to fewer lines and save a bit. Each line number takes up 6 bytes, so every line we can eliminate makes our program smaller.

Through some trial-and-error experimentation, I got this:

```
0 REM DATASTR3.BAS
10 TIMER=0:TM=TIMER
20 FOR A=0 TO 15
30 READ B$:IF B$="*" THEN 50
35 FOR I=1 TO LEN(B$) STEP 2:B=VAL("&H"+MID$(B$,I,2)):NEXT
40 NEXT
50 PRINT TIMER-TM
60 END
100
DATA000102030405060708090A0B0C0D0E0F101112131415161718191A1B
1C1D1E1F202122232425262728292
A2B2C2D2E2F303132333435363738393A3B3C3D3E3F404142434445464748
8494A4B4C4D4E4F50515253545556
5758595A5B5C5D5E5F606162636465666768696A6B6C6D6E6F7071727374
757677
107
DATA78797A7B7C7D7E7F808182838485868788898A8B8C8D8E8F90919293
9495969798999A9B9C9D9E9FA0A1A
2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5B6B7B8B9BABBBCBDBEBFC
0C1C2C3C4C5C6C7C8C9CACBCCCDCE
CFD0D1D2D3D4D5D6D7D8D9DADBDCDDDEDFE0E1E2E3E4E5E6E7E8E9EAEBEC
EDEEEF
108 DATAF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF,*
```

DATASTR2.BAS – speed **167**, size 532

As you can see, this is slightly faster than the previous combined hex string version because it does less READs. It is also slightly smaller because it has less line numbers. And, I think, it could even be packed a bit more, but because I am loading these test programs as ASCII files in to XRoar, the lines cannot exceed 249 characters (the same as typing them in) so this was as much as I could fit on them (even though using `EDIT` on these lines shows I could still type about 6 more characters, but it only seemed to show me 5 more after I re-listed it).

Fun with `DATA`, eh?

Until next time, I leave you with this:

A virtual cookie goes to the first person that finds them.

Comment 1

There's more to it than space–there's time. Depending on how strings are represented internally, concatenation may require repeated scanning of the string as it's built up. It's true for NULL-terminated strings in C, and for the Haskell String type, which is a list of characters. "Can you say O(nff2)? Sure, I knew you could."

The way around it in C is printf (or sprintf if you really need to store it rather than print it. in BASIC, if I'm just going to print it, I'll just PRINT "" (OK, &H1E if you want :). I know printf() will keep track of where it is and not repeatedly scan, and I'd bet BASIC PRINT does as well.

Comment 2

Oops… The blog software made most of the PRINT statement go away. It should have had what you were assigning to A$ in the example, but with ; instead of the +.

Comment 3

Actually, that probably isn't O(nff2), because you re-scan once for each string, not each character in each string a la bubble sort–but at the very least it ups the constant factor, which, while it's not considered significant for bigO calculations, still can be a significant performance hit. (That's why, for example, a typical sort implementation switches from quick-sort to a simpler sort once the hunks being sorted are sufficiently small.)

Comment 4

Color Basic stores strings with an 8 bit length and a pointer to the data. They're actually binary clean as a result and there's no need to scan the string to find the end. String concatenation is actually O(n) where n is the combined length of the two strings. (It has to copy both strings to a newly allocated string space.) String space allocation is O(1) unless garbage collection is triggered in which case it's O(nff2) where n is the number of extant string descriptors (which may be larger than the number of string variables due to anonymous strings being on the "string stack" during expression evaluation. The string stack overflowing is what causes "string formula too complex".).

Skipping the concatenation step and just joining with ";" (or nothing) in PRINT is faster simply because it doesn't have to bother allocating new string space and doing the concatenation. Instead, PRINT just has to look up the various strings. Both concatenation and PRINT use an incrementing pointer to traverse strings since they both process entire strings. So, in actual fact PRINT with concatenation is O(2n) and PRINT with ";" is O(n) if you keep the constant factors since the total string length is scanned twice with the concatenation option and only once with the PRINT and ";" option.

Part 7

GOSUB Revisited

In response to part 4, William Astle wrote a very nice expansion to my musings about `INKEY` and `GOTO` versus `GOSUB`. If you have been following my ramblings, I highly recommend you check out his posting. Unlike me, he actually understands what is going on behind the scenes:

Optimizing Color Basic - ON GOTO vs ON GOSUB

> Allen Huffman has been posting a series of articles on optimizing Color Basic as found on the TRS80 Color Computer. This is a response to his most recent (as of this writing) entry. Note that I'm not criticizing his article or the conclusions he reaches in it. Instead, this is intended to provide some more. One of the things he pointed out, then explained further in a comment after I didn't understand, was how the GOSUB processing works. After the GOSUB keyword is found, BASIC acquires the line number and then scans to the end of the line or the next colon. That is where RETURN will RETURN to. This sounds as one might expect, but there was a bit of weirdness I didn't "get" at first.

William demonstrated that anything after the line number is ignored, thus:

```
GOSUB 1000 I CAN TYPE STUFF HERE WITHOUT ERROR
```

…is valid. This surprised me. If you do this:

```
GOSUB 1000 I CAN TYPE STUFF HERE WITHOUT ERROR:PRINT "BACK FROM GOSUB"
```

…when the `RETURN` returns, you will see the "BACK FROM GOSUB" message printed as expected. Anything between the line number and the colon (or start of next line, whichever is found first) is ignored. William explains what is going on in his article.

This, of course, made me do some more stupid testing. First, I modified my benchmark program to run multiple tests and then average out the results. It looks like this:

```
0 REM BENCH.BAS
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 REM
40 REM PUT CODE TO BENCHMARK
50 REM HERE.
60 REM
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

Then I reran my `GOSUB` test:

```
0 REM GOSUB3.BAS
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 GOSUB100
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
100 RETURN
```

When I run this, it prints the time taken for each run, and then the average:

```
XRoar
20 FORB=0TO1000
30 GOSUB100
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
100 RETURN
OK
RUN
 0                161
 1                160
 2                158
 3                158
 4                160
 159.4
OK
```

GOSUB benchmarks.

Now for the stupid test, I added some junk after "`GOSUB 100`" and filled it up to the end of the line.

Side Note: I am loading BASIC programs in ASCII, so the program lines load in as if they were being typed in. Thus, it counts the characters "100 GOSUB " as part of it. But, as soon as you press ENTER, that line is tokenized and `GOSUB` becomes a 1-byte token (is it 1-byte?). Then you can `EDIT` the line and Xtend it and type in a few more characters. So what I show here isn't the max line size, but it is the max line size I could load in from an ASCII BASIC file. But I digress.

My line looks like this:

```
30 GOSUB100
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWX
YZ0123456789ABCDEFGHIJKLMNOPQ
RSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDE
FGHIJKLMNOPQRSTUVWXYZ01234567
89ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRST◆
```

Now when I run this, the extra "scan to the end" time causes the benchmark to show **1507**!

But who would do that? If anything, you would have a colon and real stuff after the `GOSUB`. So I tried this by changing the space after "`GOSUB 100`" to a colon and "`REM`":

```
30
GOSUB100:REMABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKL
MNOPQRSTUVWXYZ0123456789ABCDE
FGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ012
3456789ABCDEFGHIJKLMNOPQRSTUV
WXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJ
KLMNOP◆
```

Now that's a completely legitimate line. (Pretend the ABC/123 gibberish is a really long comment.)

This benchmark shows **1508**, so no real difference. When `GOSUB` is encountered, BASIC has to scan to the end of line or a colon, whichever comes first, so it should find the colon instantly, BUT, after the `RETURN` it still has to scan through that `REM` to find the next line. Thus, it's the same amount of scanning.

This is a meaningless test.

With real code, you might be doing something like this:

```
30 GOSUB 100:PRINT "BACK FROM ROUTINE"
```

Or you could have written it out as two lines:

```
30 GOSUB 100
40 PRINT "BACK FROM ROUTINE"
```

I thought the first one should be faster, since it has one less line.

And combining lines is good.

Right?

Well, in my silly example #2 above, what if I moved the `REM` to the next line, like this:

```
30 GOSUB100
40
REMABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTU
VWXYZ0123456789ABCDEFGHIJKLMN
OPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789AB
CDEFGHIJKLMNOPQRSTUVWXYZ01234
56789ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMNOPQRS
T◆
```

That's basically the same, just with an extra line number.

When I run this, I get a benchmark value of … **860**!

Look how much faster it is by moving code to a separate line! I guess we should use separate lines after all, then…

What's going on here? The key seems to be the "REM" keyword. When BASIC encounters a REM, it can just skip to the next line. That makes it faster. But, it seems to be doing something different when a REM is in the middle of a line.

It appears it is faster to NOT put REMarks after a GOSUB.

```
30 GOSUB 100:REM MOVE PLAYER UP
```

…shows **266**. This is slower than…

```
30 GOSUB 100
40 REM MOVE PLAYER UP
```

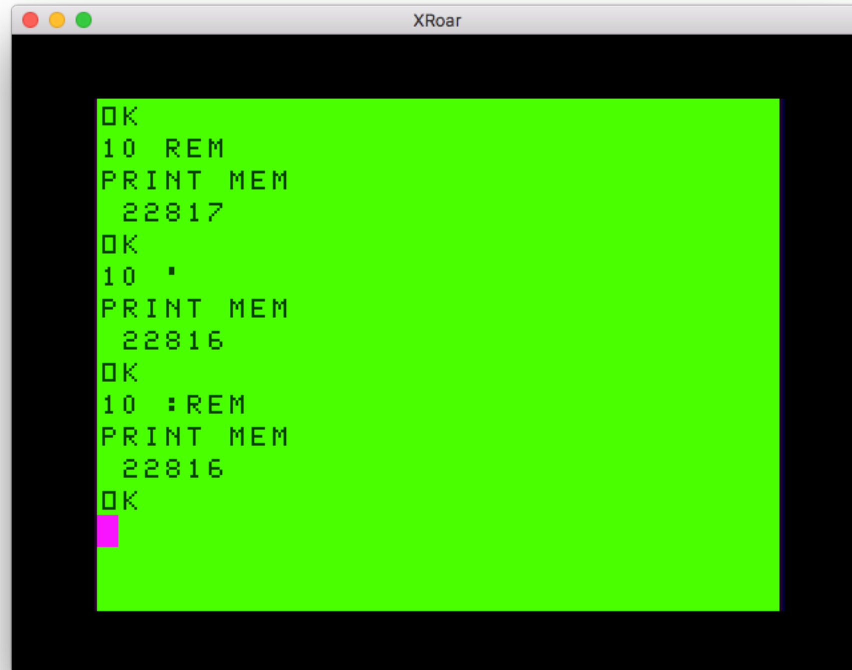
…which shows **220**. And I've certainly seen programmers make use of the apostrophe REMark shortcut.

The apostrophe represents ":REM" (colon REM) so these two are the same:

```
30 GOSUB 100:REM MOVE PLAYER UP
```

and

```
30 GOSUB 100' MOVE PLAYER UP
```



REM versus ' for comments.

Thus, using the one character apostrophe may look like it saves code space versus " :REM" but it does not. It does save printer paper, though.

But I digress…

It looks like I'm going to need another test. In the meantime, don't put things after a GOSUB om the same line. It appears to be faster to put them on the next line:

```
30 REM MOVE PLAYER UP
40 GOSUB100
```

That is **219.**

```
30 GOSUB100:REM MOVE PLAYER UP
```

That is **266**!

```
30 GOSUB100
40 REM MOVE PLAYER UP
```

That is backwards. But it produces **220**, so it doesn't penalize you for being backwards.

Oh, and as Steve Bjork pointed out in the Facebook group, a faster solution is not to use REMs at all. I think I need smarter examples. There are too many real programmers watching. For you folks:

```
0 REM THIS TAKES 371
5 DIM Z,TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 GOSUB100:Z=Z+1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
100 RETURN
0 REM THIS TAKES 357
5 DIM Z,TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 GOSUB100
40 Z=Z+1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
100 RETURN
```

Easy peasy.

A few additional REMarks

Above, I mentioned that the apostrophe represented "`:REM`". Thus, doing something like this:

```
100 'MOVE UP
```

Is slower than doing:

```
100 REM MOVE UP
```

It may look smaller, but the first example is like scanning "`:REM MOVE UP`" and the second is just "`REM MOVE UP`" so it has less work to do.

And yes, I tested it inside the benchmark program:

```
30 REM
```

…is **82**.

```
30 '
```

…is **90**.

```
30 :REM
```

…is also **90**.

I guess it's just treating the apostrophe as "`:REM`" internally, or maybe it's a 2-byte token for "`:REM`" versus a different 1-byte token just for "`REM`" or something. Dunno.

But interesting.

Until next time…

Comment 1

GOSUB and GOTO are both "two byte" tokens. Actually, "GO" is one token and then TO/SUB are separate tokens. I think this is a throwback to the original Basic which actually specified GOTO as "GO TO" (two words). By separating it into two tokens, it allows the two word variant for free due to the "skipping spaces" effect of the "next character" routine.

On the REM vs ' business: ' does have its own distinct single byte token. The tokenization process adds a : before it during tokenization behind the scenes which is why you don't need a : before ' but you do before REM. Otherwise it is treated exactly the same as REM during interpretation so :REM and ' execute at the same speed. You can see this detail by examining the tokenized basic program. Thus, if your comment is at the start of the line, you're better off using REM and if it's at the end of the line, it makes no difference.

As a bit of trivia, the tokenizer also adds a colon before ELSE which is almost certainly done as an optimization so the execution process doesn't have to look for "ELSE" as yet another "end of statement" indicator.

Even more trivial: you can replace "THEN" with "GOTO" or "GOSUB" directly. There's no benefit to doing that for GOTO, but it saves a byte with GOSUB.

Comment 2

So in the token, a : is stored before the ' token? Does this mean the detokenizer that is used for LIST has code to not print the :'s it finds before ELSE and '?

Comment 3

THEN GOTO can be replaced by either GOTO or THEN saving either 1 or two bytes. THEN GOSUB can only be replaced by GOSUB resulting in a 1 byte saving. The CRUNCH program replaces THEN GOTO with THEN, and THEN GOSUB with GOSUB. (It doesn't replace either if there is a space between THEN and either GOTO or GOSUB.) *Note* GOTO can be expressed as GO TO and GOSUB can be expressed as GO SUB.

Comment 4

Why do you not do it if there is a space? You mean like:

```
IFA>1THEN GOSUB 100
```

Part 8

Arrays and Variable Length

In part 3, I demonstrated a simple "game" where you could move a character around the screen and try to avoid running into enemies. The original version hard coded four enemies, each with their own variable. It looked like this:

```
0 REM GAME.BAS
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=256+16
15 E1=32:E2=63:E3=448:E4=479
20
PRINT@P,"*";:PRINT@E1,"X";:PRINT@E2,"X";:PRINT@E3,"X";:PRINT
@E4,"X";
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$):IF LN=0 THEN 30
45 PRINT@P," ";
50 ONLN GOSUB100,200,300,400
60 IF P=E1 OR P=E2 OR P=E3 OR P=E4 THEN 90
80 GOTO 20
90 PRINT@267,"GAME OVER!":END
100 IF P>31 THEN P=P-32
110 RETURN
200 IF P<479 THEN P=P+32:RETURN
210 RETURN
300 IF P>0 THEN P=P-1
310 RETURN
400 IF P<510 THEN P=P+1
410 RETURN
```

I then modified it to use an array for the enemy variables, so less code was needed to cycle through them, while also allowing easy changing of the amount of enemies. It looked like this:

```
0 REM GAME2.BAS
1 EN=10-1 'ENEMIES
2 DIM E(EN)
5 KB$=CHR$(94)+CHR$(10)+CHR$(8)+CHR$(9)
10 CLS:P=256+16
15 FOR A=0 TO EN:E(A)=RND(510):NEXT
20 PRINT@P,"*";
25 FOR A=0 TO EN:PRINT@E(A),"X";:NEXT
30 A$=INKEY$:IF A$="" THEN 30
40 LN=INSTR(KB$,A$):IF LN=0 THEN 30
45 PRINT@P," ";
50 ONLN GOSUB100,200,300,400
60 FOR A=0 TO EN:IF P=E(A) THEN 90 ELSE NEXT
80 GOTO 20
90 PRINT@267,"GAME OVER!":END
100 IF P>31 THEN P=P-32
110 RETURN
200 IF P<479 THEN P=P+32:RETURN
210 RETURN
300 IF P>0 THEN P=P-1
310 RETURN
400 IF P<510 THEN P=P+1
410 RETURN
```

Arrays are an easy way to reduce code. What I did not realize is how slow they are! Consider this example:

```
4 DIM E1,E2,E3,E4
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 E1=1:E2=1:E3=1:E4=1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

In the test loop we simply set four different variables. Notice that the ones we use most often (inside the test loop) I have declared first in line 4. This makes them faster, since they are found earlier when BASIC looks up the variables.

This results in **576**.

Instead of using four separate variables, we could use an array of four elements:

```
4 DIM E(3)
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 FORC=0TO3:E(C)=1:NEXT
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

Although this simplifies the code, and allows us to easily change it from 3 to more variables, it adds another `FOR`/`NEXT` loop.

The speed drops to **1408**! And that's using a one-character variable name. It might be a tad slower if I had chosen "EN" for the array or something two-characters to match the original.

It seems that, if you can get away with it, manually handling separate variables is much faster.

Arrays will win for code size, but lose for speed. I probably won't want to use arrays to track the enemies in my BASIC arcade action games.

Can we make arrays faster? Let's remove the `FOR`/`NEXT` loop and access them manually, so it's as direct a compare to non-arrays as we can get:

```
4 DIM E(3)
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 E(0)=1:E(1)=1:E(2)=1:E(3)=1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

Now we are doing "`E(0)=1`" versus "`E1=1`". Arrays should still be slower because there are more characters to parse, and an array lookup as to be done.

This produces **1021**. It appears that setting the variable takes about a third of the time, looking up the array another third, and the `FOR`/`NEXT` loop a third third. Or something.

As brute-force as it looks, it appears that is the faster way for handling variables even though it produces more code.

And speaking of more code…

Variable Length

One-character variable names can get confusing real quick, but the two-character limit in Color BASIC isn't much better. Well, you can specify longer variable names, but only the first two characters are honored:

```
USERNUM=1
```

…turns in to…

```
US=1
```

If you could ensure the first two characters were unique, you could make your program more readable, but you would be wasting speed and code size.

Consider this benchmark example, which uses a 10-character variable:

```
0 REM VARLEN.BAS '211
4 DIM USERCOUNT
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 USERCOUNT=1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

This produces **211**. It's wasteful, since BASIC is only honoring the first two characters. Let's try this:

```
0 REM VARLEN.BAS '182
4 DIM US
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 US=1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

This produces **182**. All those extra useless characters did nothing but slow things down a tad.

But using a one-character variable is the fastest we can get:

```
0 REM VARLEN.BAS '177
4 DIM U
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 U=1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

This produces **177**.

For the most-used variables, use a one-character variable name for the best speed.

And remember, every time a variable is references, BASIC has to start with the first variable it knows about and walk through all of them until it finds a match. That is the purpose of the `DIM` statements in lines 4 and 4. They are declaring variables in the priority of most-used to least, sorta. The variable used in the inner `FOR`/`NEXT` loop is U, so I declare it first.

If I had done U last:

```
0 REM VARLEN.BAS '177
6 DIM TE,TM,B,A,TT
7 DIM U
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 U=1
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

This slows it down to **188**.

Putting it all together: Avoid arrays, use one-character variable names, and variables you want to be the fastest should be declared earlier.

Just an FYI.

Comment 1

It actually follows that array accesses would be slower. Unsurprisingly, it has to look up the array itself, though that isn't particularly slow since arrays have their own table that lives just after the regular variable table. Of course, if you have multiple arrays, the first array defined will be found more quickly than the tenth since it's still a linear scan of the array table.

Incidentally, it's better to allocate all your regular variables and then dimension any arrays for just that reason – the arrays have to be relocated any time a new scalar variable is allocated. That can substantially slow down variable allocation. Thus, you should have your array defining DIM appear after the scalar defining DIM.

Anyway, in addition to just looking up the array, the subscript has to be calculated which involves a trip through the expression evaluator which could lead to any number of things like variable look-ups, etc, but which will at a minimum be either a variable lookup or a constant conversion. Then the subscript is converted to an integer and then bounds checked.

Once the subscript is determined, it is a simple calculation to get the offset into the array data to find the actual value, but it is a calculation that is not required for scalars.

Note that for multidimensional arrays, you get to repeat the subscript lookup multiple times so they will be correspondingly slower. Theoretically you could have up to 255 dimensions though you would be hard pressed to actually do anything useful with that.

If you think about it, array type accesses are slower compared to a direct variable access even in low level languages like assembly language. It's unavoidable since additional work is required to do the offsetting into the data.

Comment 2

Just as side note, a story about array usage on a BASIC dialect with same ancestry, Commodore BASIC V2 on a C64. For a program doing Hires graphics only in plain BASIC we tried to accelerate the pixel access by using a integer array (16 bit value, alas, not implemented in Color BASIC). Some would expect, the clumsy address calculation into the bitmap could be boosted using array access, but this failed. It was slower! It seemed elegant, but the index calculation (expression evaluation as mentioned above) took its toll, beside the fact that the adjustment calculation to signed 16-bit value to pixel location was clumsy, too.

Comment 3

I actually plan to use VICE and try to do similar benchmarks on a C64 (it has TIME) and see how similar they are. Is there a TIME on the VIC-20?

Comment 4

Yes, it's the same BASIC version.

Comment 5

They appear to use the 60Hz screen refresh like the CoCo. I assume there were 50Hz PAL versions, too?

Comment 6

No, the BASIC timer is alway 60Hz even on PAL machines. It is based on the hardware timer which starting value is set accordingly to the base frequency of the system. It's not derived from the raster interrupt.

Comment 7

+1 for Commodore! I think the CoCo's timing is based on the screen refresh, so programs that relied on it would have to ask the user if they were playing on a 50Hz or 60Hz machine :)

Do you know of an online C= emulator? I think I found a C64 JavaScript one, and I'd like to find a VIC-20 one as well.

Comment 8

> +1 for Commodore! I think the CoCo's timing is based on the screen refresh, so programs that relied on it would have to ask the user if they were playing on a 50Hz or 60Hz machine :)

Hmm, just partially +1, let's say better +0.5, because on a C128 with BASIC 7.0 additional sound(!) and graphic commands which are timing related are based on the raster interrupt timing. Not a bad idea for graphics, but sound!? Therefore, BASIC 7.0 programs has to ported or carefully coded to support NTSC and/or PAL … :(

> Do you know of an online Commodore emulator? I think I found a C64 JavaScript one, and I'd like to find a VIC-20 one as well.

Just know of one in Java, see http://www.z64k.com ….

Comment 9

Very interesting! I understand why sound would be (duration of notes), but what did graphics have to do with it?

Comment 10

Raster interrupt timing hits graphics in multiple ways. First, the mixed modes (graphic and text-mode on the same screen) depends on raster timing. Second, controlling sprites with command MOVSPR allows you to let automatically move a sprite over the screen with a certain direction (angle) and speed. The latter is based on raster interrupt's screen refresh rate. Beside this, the official commodore manuals are horrible, containing many more or less severe errors or just missing information regarding this issue (at least in the German translation).

Comment 11

I was unaware of Sprites through BASIC, and split modes. Very nice. A friend of mine had a C128, and I recall playing with the basic graphics commands (seeing what it had compared to the CoCo's EXTENDED BASIC commands). I had a Commodore friend who introduced me to GEOS, and we went to another C128 user's house to use the 128 GEOS and some really NICE desktop publishing program to make tickets and fliers for a rock and roll band thing I was helping with. There was some impressive non-game stuff for the 128.

Was the TI$ different, then?

Comment 12

Nice use-case of a C128 … another kind of “band aid”.

I appreciate any non-game stuff.

TI$ is always 60Hz. It’s coming from the depth of the Kernel which provides the timer function. BASIC only maps it into pseudo variables.

All Commodores do it the same way, with one exception: the B-series (AKA Commodore-II, PET successor) has no pseudo variable TI, only TI$, but with 7 digits instead of 6, carrying an additional 1/10th second digit. But this Commodore model branch didn’t play a role, so compatibility to them wasn’t an issue.

Comment 13

> Putting it all together: Avoid arrays, use one-character variable names, and variables you want to be the fastest should be declared earlier.

In addition to that (I think this fits well here), I would also suggest to prevent constants in preference of using variables. Even the variable name space is quite limited (26+26*36), a value already in floating-point representation store in a variable -assuming it is defined “early” as visualized above- could be faster than the parsing and conversion of constants (involving a trimmed by 10 float multiplication for each additional digit). Caveat: Variables don’t make it for all constants, there is a break-even point between number of digits and the position in the variable table. If I correctly remember, for a 6502-based MS BASIC one digit constant is faster then a single-char variable if its table position is greater than 11, which may probably vary for Color BASIC. But this might be a subject for further investigation and another posting in this interesting article series Allen will bring to us eventually …

Comment 14

There’s always READ/DATA, but the global nature of Color BASIC, along with not having RESTORE like BASIC09, would make it obnoxious. (Not to mention that I was kind of bummed that neither BASIC would let you READ and read the individual elements for you. Drat.)

Comment 15

I should know better than to type BNF-ish things here. Pretend I typed “RESTORE [line number]” and “READ [array name]”. No RESTORE??? Ugh.

Comment 16

RESTORE is there. You just can’t have it reset the pointer to a particular line number – only to the start of the program. The kicker is that it wouldn’t be terribly difficult to implement.

I can see that READ [arrayname] could be useful. However, that is something of quite limited use but would require some special (read as duplicating existing variable parsing and lookup code) coding in the interpreter. Given that there is a straight forward alternative for the relatively few cases where it would be useful, it’s not surprising it isn’t there. It’s better to use the code space for other more useful things.

Now, if anyone can explain to me how the ROM code for READ and INPUT (it’s the same routine, believe it or not) actually works from studying it, I’ll be seriously impressed. That’s probably the most obscure code in the entire interpreter save for the floating point code. Adding a “fill array” option for READ would be seriously hard given the code that’s already there.

Comment 17

Color BASIC has RESTORE; it doesn’t have the option of a line number following RESTORE to let you set it someplace other than the beginning.

Part 9

Have no fear! Today's installment is a short one. It will address a few miscellaneous things I have been told about.

"." versus "0"

In a comment posted to Part 5, **Darren Atkinson** (designer of the fabulous CoCoSDC interface) pointed out a place where a decimal point makes things faster!

> One place where a decimal point will give an improvement is when using the value 0 in an expression. Basic will accept a stand-alone decimal point as the number 0, but it will process it faster than the '0' character.
>
> Try comparing the speed of:
>
> ```
> IF N < 0 THEN ...
> ```
>
> with that of:
>
> ```
> IF N < . THEN ...
> ```

I, of course, had to test this. Using the benchmark program:

```
0 DIM Z
5 DIM TE,TM,B,A,TT
10 FORA=0TO4:TIMER=0:TM=TIMER
20 FORB=0TO1000
30 Z=0
70 NEXT:TE=TIMER-TM
80 TT=TT+TE:PRINTA,TE
90 NEXT:PRINTTT/A:END
```

Setting Z=0 1000 times produced the value of **178**.

```
30 Z=&H0
```

Using a hexadecimal zero produced **164**.

```
30 Z=.
```

And using just a period to represent zero produces … **141**!

Okay, no more zeros.

Fake FOR

George Phillips chimed in about `GOTO` and `GOSUB` with a sneaky way to jump around faster:

However, because BASIC stores lines as a singly linked list the fastest places to GOTO and GOSUB are either the top of the program or anywhere after you do the GOTO/GOSUB. BASIC is clever enough to look forward if the line # is after the current one, otherwise it must search from the top.

Using a "fake" FOR/NEXT loop instead of GOTO could well be faster in such cases since it doesn't have to search for the start of the FOR. If you have:

… whole bunch of code

```
1000 PRINT"HERE WE GO."
```

… do some stuff

```
2000 GOTO1000
```

It is likely faster to do this:

```
1000 FORX=0TO1STEP0:PRINT"HERE WE GO."
2000 NEXT
```

Possibly tricky to use in practice, but you can have a lot of FOR/NEXT loops on the stack and skip over interior ones. Fairly unpleasant, but optimizing BASIC is not a pretty undertaking.

Wow. James Gerrie and Johann Klasek also mentioned this in response to Part 4. Johann gave an example:

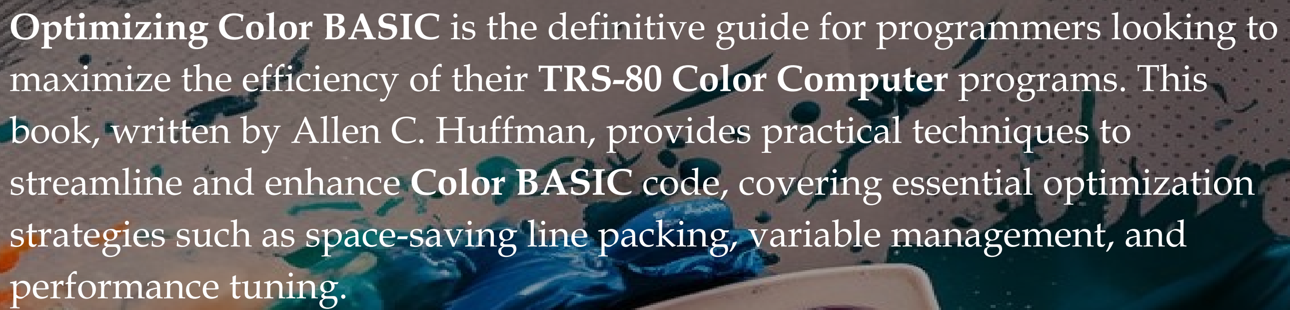
I think something like that was in mind:

```
10 FOR A=. TO 1: A=. : REM FOREVER
20 ON INSTR(" UDLR",INKEY$) GOTO 100,200,300,400,500
30 REM FALL THROUGH ACTION
40 NEXT
50 END
100 REM IDLE LOOP
110 NEXT
200 REM MOVE UP
210 NEXT
300 REM MOVE DOWN
310 NEXT
400 REM MOVE LEFT
410 NEXT
500 REM MOVE RIGHT
510 NEXT
```

Compared to the ON GOSUB this is some kind of "redo" or "loop retry" not reaching the fall through action in 35.

Anytime the overhead of scanning through lines (from first line to destination) is more than the overhead of a `RETURN`, this would be faster (and only use a bit of extra memory for remembering where to `RETURN` to).

I don't have any benchmarks for this one, but there is probably a threshold where the number of lines before the `GOTO` has to be more than X before this is always faster. Thanks, Darren, James and Johann (and any others I might have missed).



Optimizing Color BASIC is the definitive guide for programmers looking to maximize the efficiency of their **TRS-80 Color Computer** programs. This book, written by Allen C. Huffman, provides practical techniques to streamline and enhance **Color BASIC** code, covering essential optimization strategies such as space-saving line packing, variable management, and performance tuning.

Whether you're a hobbyist or a seasoned retro-computing enthusiast, you'll discover invaluable insights into:

- Removing unnecessary spaces, REM statements, and redundant variables to boost speed.
- Optimizing loops and GOSUB calls for efficient execution.
- Using hexadecimal values and memory management techniques to reduce processing time.
- Enhancing keyboard input handling and display routines for faster response.
- Advanced topics include leveraging inline assembly and rethinking algorithm structures for better performance.

With step-by-step examples, benchmarks, and practical tips, **Optimizing Color BASIC** is an essential resource for anyone looking to push the limits of the Color Computer's capabilities. Unlock the full potential of your programs and experience Color BASIC like never before!

Radio Shack

TRS-80 Extended Color BASIC System Software;
@ 1984 Tandy Corporation and Microsoft.
All Rights Reserved.