The QB64 Bible

By Steve McNeill

smcneill@swva.net

# Chapter 01: Introduction

 Hello World!! Welcome to my version of ***THE*** reference book for all things related to QB64. “Who am I?”, you ask? “What qualifications do ***I*** have to write a reference book on QB64 and call ***The QB64 Bible***?”

 Excellent questions! Let me answer those for you, and then tell you a bit about what you’re reading here and what this reference book is intended to accomplish.

 First, to start with myself, I’m Steven McNeill, from a small little place called Pilot, Virginia (in the great ole US of A for those who don’t know), and I’ve been working and coding in various forms of the BASIC language since I got my first computer sometime around 1980 or so. (As of this moment, the year is 2022, so that’s 42 years of programming in one form of BASIC or another.) I’d think that longevity with the product makes me qualified to write on its usage, based on ***experience*** with the BASIC language itself.

 As far as my personal experience with QB64, I first stumbled upon the project probably sometime around 10 years, or so, ago, back when Galleon was still the only developer for the language. At that point, the version would’ve probable been something around v0.7, or so, and I instantly fell in love with the nostalgic IDE and how similar the product looked to both QBASIC and QB45 – languages that I had studied back in high school in the early 90s, and which I knew and favored extensively. QB64 was like a small slice of Heaven to me, bringing back a lot of memories and ease of coding short, quick programs back to me.

 After several months of learning the new underscore commands, and familiarizing myself with what the language could do, I decided to delve into the internal structure of the language and try and uncover what the secret to Galleon’s magic was. Translating BAS code to C code and then compiling it into an EXE that can run on modern systems? ***ASTOUNDING!!***

 At first, the internals were a complete and utter mess of GOTOs, tiny undescriptive variable names, and an utter headache to try and sort anything out. Eventually, I sorted out that the compiler runs code in multiple passes – a prepass to sort out constants and types and such, and then a main pass to translate the code from BASIC to C. At this point, I finally started to modify the internals of QB64 itself and added code into the language to help expand what it was possible for CONST to do for us. This was around version 0.9, or so, and was also the time when Galleon welcomed me onto the team as an ***Official Developer*** for the language!

I hope that recognition alone would be enough to help others recognize my ***expertise*** with the language. Over time, I added even more features into the language, squashed a ton of bugs, and helped expand what we can do with the language.

I’ve got years of ***experience*** with the language. I’ve got ***expertise*** with the language. And, I’m a professional writer by trade now, with the overwhelming ***ego*** to take time to sit down and write up a guide which I hope will become ***the definitive Bible for QB64.***

If not me, then ***who?*** ;P

Now, with all that said and out of the way, let me take a moment to address, “***Just what the heck am I hoping to accomplish, and how am I going to accomplish my goals?***”, with this reference material.

What I’m hoping to accomplish is to simply write a definitive reference tool that all coders in QB64 would love to have a copy of on their bookshelf, or saved as an e-file somewhere on their drive.

How I’m going to do this, is simply by trying to share all the information which I’ve gathered on QB64 over the years, and attempt to present it in a manner where somewhere can reference that material and make use of it. The exact format, at this moment in time, may start out as a crude jumble of disjointed information, but as I write and expand this guide, I hope to fill in any blanks which users might have and, with their feedback and suggestions, organize my thoughts into the most efficient way with which others can make use of them.

For the moment, I’m going to start with one single subject or command, and then try to describe the nuances and usage of that command to the best of my ability, and then from their expand outward to cover as much of the language as I possibly can.

And, with that too said and done, let’s get started with this project!

# Chapter 02: Numbers and Numeric Types

 Originally, I’d started this reference by opening up with Screen and Console as my original topics to break down and discuss. Unfortunately, however, I soon came to realize that I needed to start with something a little more fundamental to begin with, as its basic concept permeates every aspect of QB64’s existence – and that’s numbers and numeric types!

 To start talking about numeric types in QB64, I’m going to begin by introducing you guys to a numeric type which ***isn’t*** supported by any programming language I know – FINGERS!! Now, since I assume everyone reading this is a human being, the concept of counting via fingers shouldn’t be one too foreign to everyone. (Everyone reading this ***is***  human right? Or at least human-adjacent?)

 For those who might not be human, or for those who aren’t familiar with the concept of counting via fingers, I’ll take a brief moment to walk everyone through the process.



 First, say the word “ONE”, and hold up your left hand with the finger near the thumb pointing up. It should look like the leftmost hand and finger in the picture to the right.

 Then, say the word “TWO”, and hold up the middle finger along with the index finger.

 Now, surely I don’t have to insult anyone’s intelligence to continue this sequence of counting, as it’s generally something that human beings learn to do some time around preschool, or right as they enter school. From the picture above, we have a visual representation for the values of one, two, three, four, and five.

 But, now here’s where it gets a little tricky… What happens when we have to count beyond five??

 *“We use the other hand!”*

 Yes, Mr. Smartass, I hear you answering in your snarky little know-it-all voice! Then what happens when we count to ten, in that situation?

 *“Umm…. We run out of numbers!”*

 Right! Well, wrong, but you could be right! At least on a fundamental level of things.

 But, before I explain how you ***could*** be right by saying, “We ran out of numbers,” I want to take you guys back to your preschool days and ask you the same question again: “What happens when you add one to ten?”

 At first, your young little mind pauses. You frown heavily and try and work out a solution. You may be a gifted youngster and know that eleven comes after ten, but how do you \****show\**** that value with your fingers? Surely everyone reading this has seen a young child struggle with this dilemma at least once in their life, right?

 Usually, they go for the simplest of solutions: They hold up all ten fingers, then pause for a second, show you no fingers, and then hold up one the index finger of their left hand as they proudly say, “Eleven!”

 Ask them to show you how many thirty is, they’ll show ten fingers, close them, show ten fingers, close them, show ten fingers, and then proudly proclaim, “Thirty!”

 This is ***OVERFLOW MATHEMATICS***!! You’ve ran out of digits to represent your value, so you simply start over from the beginning again. When all is said and done, and all the fingers are opened and closed fully, what you’re left holding up for thirteen is the same the three fingers you hold up for three, and the same fingers you’d end up holding up for twenty-three, thirty-three, forty-three, and so on. The numbers aren’t actually the same, but we don’t have enough hands to hold all the values we might want, so we overflow by opening and closing our hands until we’re left with a value which we ***can*** represent with our fingers.

 And since everyone should be able to understand this very simple little human concept, then they shouldn’t have any issues when I tell them, “***Your QB64 programs use this exact same overflow mathematics!***”

 If you dim a variable as an unsigned byte, it can only holds values from 0 to 255. What happens when we try and give it a value of 256??

 As programmers, we’re left with two choices really in this case:

 Choice one: Be as mentioned above and simply decide, “We’ve ran out of numbers!” Toss an Overflow Error, stop program execution, and prevent the program from moving on and doing anything else as we’ve tossed it numbers that are simply out of bounds for it.

 OR, Choice two: Do as small children do when representing numbers that are larger than the amount of fingers they have – overflow the values and only keep the digits that we can actually represent at the end of the process.

 QB45 and QBasic chose to go with choice one. They toss Overflow Errors and basically stop program execution at the point where you exceed a number’s limits.

For QB64, Galleon (our original developer) asked the community how they wanted to handle this issue. We chose to allow for overflow to occur, accepting it as the lessor of two evils, rather than have a program completely halt and stop on us.

For QB64, if you put a value of 256 in an unsigned byte, and then print it to the screen, you’ll see that the value it actually contains is 0. ZERO. You went beyond the limits of the variable type, overflowed, and was left with a big fat zero for your effort. If you put a value of 257 into an unsigned byte, it’ll end up storing the value of 1 instead, as you overflowed the register.

QB64 doesn’t throw automatic, program-ending, errors when you exceed a variable’s limits. Instead, it quietly overflows the value and only holds the remainder. This can lead to some obviously unintended consequences if someone isn’t prepared for this overflow. (Of course, having your program crash and end, screaming “OVERFLOW ERROR”, isn’t any better of a solution to the issue, now is it?)

One example of this overflow doing unintended things would be like in this example below:

|  |
| --- |
| Dim A As \_BYTE |
| DO |
|  A = A + 1 |
|  PRINT A |
| LOOP UNTIL A = 128 |

Now here, we have a simple little DO… LOOP that counts from 0 to 128… Or, it ***should*** count from 0 to 128. Unfortunately, we assigned our variable A as being a simple byte. Bytes can only hold values from -128 to 127. 128 is beyond the storage scope of a byte, so it’s going to overflow, leaving us a value of -128, in this case.

From the lowest value we can hold, to the highest value we can hold, overflowing back to the lowest value we can hold once again – for those who might ask, “Why is the overflow value -128?” -128 is the lowest value which a byte can hold, so when we overflow past 127, we start counting from that low point upwards again. 128, stored inside a signed byte, ends up becoming the exact same value as -127 after it overflows beyond it’s normal bounds.

Which is why, for the little snippet of code I posted above, the program ends up in an endless loop. A starts at a value of zero. It then enters the loop and increments to 1, where we print it. The program sees that A isn’t equal to 128, so it loops back. Increments, prints 2. Check, loop, increment, print. Repeat multiple times….

Finally, A holds a value of 127… We try to increment it and add one, overflowing that a byte can hold. At this point, A now ends up holding a value of -128.

***127 + 1 = -128***

Only when dealing with computers could anyone hold the above math to be valid, but it’s what we get when dealing with overflowing addition with bytes! If our variable type had been an integer, this math would, of course, been perfectly valid and never overflowed. Of course, with integers, we ***still*** have a limit for the values which they can represent – from -32,768 to 32,767. The moment we try and cram a value of 32768 into that integer, is the moment it’ll end up overflowing back to -32768!

It's the same principal at work, just with larger limits.

***When you try and store a value larger than what a variable type can hold, that value is going to overflow and all you’re going to store is the remainder in that variable.*** Everyone needs to be aware of this basic rule of numbers and numeric types when programming in QB64.

## Chapter 02.01: \_BIT Types

Since we’ve already spoken a bit about overflow values and how they behave, let me take a moment to speak about something else that a lot of folks seem to have a misconception over – the \_BIT variable type.

A bit is simply a single on/off value. Signed, a bit will either be -1 or 0. An unsigned bit will either be 1 or 0. This is an immutable fact, and it’s one that everybody who has dealt with any sort of serious programming already knows. A bit is nothing more than a way to tell if something is ON or OFF.

So, what’s the problem that people have with the \_BIT type, you ask?

It seems a lot of folks think that by using \_BIT, they’re using less memory and making their program more efficient and faster.

They’re WRONG!

By itself, a \_BIT variable type is simply going to slow down your program and make things less efficient. Honestly, the use of \_BIT and \_UNSIGNED \_BIT should be very limited and only done in one singular situation – which I’ll go into below.

“What?! How’s that possible,” you ask?

Sometimes it helps to step back away from computers and talk about things in a different manner that folks can relate to easier and more visually, and that’s what I’m going to do here to help explain the concept. Let’s take a moment and talk about cars. We park cars in our garages, driveways, or in specific areas outside our house. When we need to go somewhere, we hop in those cars and drive from where they were parked at, to somewhere close to our destination and then we park them there and get out. For various purposes, we have different sized cars – from a sedan which holds a single family, to a van which can seat a small school team, to a bus which can seat multiple people.

Now, thinking in the same terms as above, let me explain our variable types:

A \_BYTE would be the single family sedan. A small car which can hold the smallest group of people.

An INTEGER would be the full sized van. You can get several people in it.

A LONG is the bus. It can hold a whole crapload of people in it.

“Now, where’s the \_BIT,” you ask?

It’s the PEOPLE!!

A sedan can hold a family – a byte can hold 8 bits. A van can hold twice as many – an integer can hold 16 bits. The bus can hold a whole swarm of people – a long can hold 32 bits.

If we compare variable types to cars, then we have to say that \_BIT is comparable to the people who sit in the cars.

“And how does this make our programs more inefficient or slower to run,” you ask?

Keep thinking in terms of the cars. Whenever you want to leave your house and go somewhere, like to work, what do you have to do first? You have to hop in a car to drive there!

You don’t see people walking down the middle of the interstate, and you don’t find singular bits being added or subtracted in your computer’s registers. \*ALL\* people have to be in a car to drive, and \*ALL\* bits have to be in a byte to stay and travel in memory.

When you define a variable type as a bit, you’re \*STILL\* using a byte of memory. All you’re doing, if you want to think of it in terms of the car analogy above, is telling your insurance company, “I only want to certify this vehicle as a single occupancy vehicle.” You still have the same car as before (or the same byte in memory); you’re just pledging that you’re only going to make use of a small portion of it. For the car, you don’t magically shrink to a compact car – you still have the same car using the same amount of space and gas on the road – you’re just saying that you’re always going to keep the passenger seat and rear seats empty.

It's the exact same way with a \_BIT – you’re just saying that you’re always going to keep the rest of your byte empty and never make use of it.

“Well, that’s fine.” I can hear you guys thinking that already, along with, “Now how the heck does that make my program any less efficient and slower?? All it’s doing is limiting what I’m going to put into a byte!”

 That’s due to the simple nature of computers! Take just a brief moment to think about what’s going on behind the scenes, and you’ll quickly see why the use of \_BIT is slower than not using it…

(Giving you a moment to think on your own…)

Got it? Maybe not? Then let me just tell you guys the answer, with a question!

Which is faster – a car that just drives down the road, or a car where you have to wait for the driver to get into it, before driving down the road??

If you define a variable type as a \_BIT, you’re basically declaring it to \*only\* be a portion of a byte. To make use of it, the computer has to first look up that byte in memory, and ***then look only at the portion of it that’s the bit!***

Instead of just saying, “Look at that Volkswagon going down the road,” it has to specify, “Look at ***the driver*** of that Volkswagon going down the road!”

A program that looks at a byte, then looks at a bit, is obviously going to be a little slower than a program which simply looks at a byte. It’s got an extra step to process built into it!

So, let’s take a moment to review what we know from this:

1. All variables dimmed as \_BIT are going to use at least a \_BYTE in memory.
2. Accessing and writing to the contents of a \_BIT is going to inherently be slower than working with a \_BYTE.

So, what do we gain by making use of a \_BIT?

Absolutely squat, except for the fact that we make certain our variable will only overflow between two states. As long as you’re not counting on having to force the variable to purposefully overflow, ***there’s absolutely no reason to define a variable as \_BIT***.

“But above, you said there was a singular case where \_BIT might be acceptable!”

I did, and that singular case is when dealing with an array and bit packing!

DIM x AS \_BIT – This little line of code will use a byte of memory (the smallest size possible) to store that single bit in it.

DIM x(0 TO 7) AS \_BIT – This little line of code packs the bits together and still uses a single byte of memory.

Using a \_BIT type makes sense if you’re wanting to pack an array and reduce memory usage – but that’s they only time when using \_BIT makes any real sense. AND, even then, it only makes sense as long as you realize that it’s going to slow down your program every time it accesses one of those bits. After all, it’ll still have to both look up the byte that the bit is stored in, in memory, and then it’ll have to look at that specific bit.

If your program is running into memory issues, and you have to keep memory usage down – and if speed isn’t a concern for your program – then you might consider making use of a \_BIT array.

Otherwise??

Just forget the hassle and use \_BYTE instead!

# Chapter 03: SCREEN and CONSOLE

 What could possibly be a better place to start when talking about a programming language, than in talking about the various screen modes and windows which it has access to? Many people start with a simple “HELLO WORLD” program, but I’m hoping that anyone who reads this reference would already know how to do something so simple. If not, then here’s a quick spoiler for you:

PRINT “HELLO WORLD”

 Copy the above code into your IDE, hit F5, compile and run! Yaaaayyyy!! That’s a quick little Hello World program!

 But, ***where did that Hello World end up displaying??***

 The compiled program looks like it compiled and ran on a standard console terminal, but that’s not what happened. If you watch *very closely* as the program starts up and runs, you’ll notice that the screen first pops up in one spot on your computer, and then it disappears and reappears elsewhere, where it stays unless you manually grab it and move it like you can any other window. This is because your program actually starts in a very small console window, and then Open GL (the graphics engine upon which everything is built and runs behind the scenes) takes over and creates a graphical window – which \****emulates\**** a console window!

 Sounds counter-intuitive, doesn’t it? QB64’s default screen mode ***isn’t*** a console window, but is instead a graphical emulation of a console window!!

 As odd as it sounds, this is how things work, and it’s also been an issue with many programs which folks have shared over the forums and chat channels over the years. Be aware of this behavior, as it could come back to cause issues in your work in the future, as your programs get more complex and robust.

The biggest place where I’ve generally seen this swap from console to graphical window cause issues, is when people want to do something with the window handle at start up, such as removing the borders or title bar from the window. What happens is they end up getting the window handle (the reference for where the window exists in memory) for the initial console window, which then closes as the graphical window is created, and then they try and use that invalid and obsolete handle to manipulate their program window – which, of course, won’t work!

Generally speaking, adding a very short delay to the start of your program (even a \_DELAY 0.2 is probably large enough) is enough to prevent any issues as it gives your operating system time to close the console window and create and register the graphical window, so it’s not usually a very hard issue to fix and address – once someone knows it’s something they may have to address. If you don’t know it’s an issue, and don’t have any clue that it can be a problem, then how the heck can you ever sort out how to fix the problem??

Which is why I wanted to go through the trouble of mentioning what’s going on behind the scenes, and to make people aware of the possible issues first and foremost, before going into any real details about our various screen modes and console uses.

## Chapter 03.01: Screen Modes

 QB64 has several screen modes which it can support, but I’m only going to ever address three of them – SCREEN 0, SCREEN \_NEWIMAGE(x, y, 256), and SCREEN \_NEWIMAGE(x, y, 32). All the rest of the modes are legacy modes which were used on old computers (We’re talking EGA and VGA graphic modes – which you younger programmers will probably have to google and look up to even understand those abbreviations.), and offer such tiny screen resolutions that on today’s monitors and equipment, the screens are almost impossible to read and interact with.

 For example, here’s the limitations of an old SCREEN 7 program.

 Screen Width: 320 pixels

 Screen Height: 200 pixels

 Screen Colors: Max of 16 colors

 Now, let’s compare that to what a normal monitor and program offers nowadays:

 Screen Width: 1920

 Screen Height: 1080

 Screen Colors: 4,296,967,296 colors.

 Now, this is looking at a 1080p HD monitor – which is actually rather low-end on the equipment range for modern machines. 4K displays run at 4096x 2160 pixels, so you can easily see how small of an area these old programs would use. Or, if you can’t image it, I’ll illustrate one for you:



 Now, see that red square in the top right corner of my screenshot? That’s 320x200 pixels on a 1920x1080 screen! Can you imagine writing a program that uses that little of the screen – and remember, the lettering on this screen has to be small enough to fit 25 rows of text onto it! Think you can read that? I sure as hell couldn’t!

 And, even worse, a 4k display device is going to be ***less than*** ***half*** that height and width!!

 ***Modern devices don’t work well with 1980 era limitations!*** Our technology has progressed. Pixel size has shrunk. Monitors are the same size as they ever used to be (a 13-inch laptop ***still*** has a 13 inch display ), but the resolutions on those devices are greater than ever and are only going to increase as the future progresses.

 Everyone I know who has written or produced code for these outdated screen modes always ends up asking, “How can I make the screen bigger so I can actually see it?” Now, the answer to that varies and there’s a lot of easy solutions we offer, but I’m not going to go into those. I personally see no reason to teach folks how to do something that is only going to be a problem for them in the future. If you absolutely must use these outdated screen modes (Mode 1 to Mode 13), you’ll need to be self-reliant enough to troubleshoot and learn those modes on your own. They’re there, but they’re obsoleted and only for legacy programs, and I’m not going to waste time and pages detailing and documenting each of them.

 All I’m going to deal with in this book, and all I’ll ever work with, are the following:

 SCREEN 0 – a text only mode.

 SCREEN \_NEWIMAGE(x, y, 256) – a 256 color mode, whose use is currently limited by QB64’s inability to load 256 color images, which defeats the whole purpose of the mode.

 SCREEN \_NEWIMAGE(x, y, 32) – a 32-bit color mode, which honestly should be the standard which everyone uses for modern graphical programming.

 CONSOLE – the actual screen terminal/console window. Personally, I find this to be an invaluable debugging tool, and I strongly believe that all programmers need to know how to make use of it.

## Chapter 03.02: Screen 0

Of the screen modes that I’m going to be talking about, SCREEN 0 is one of the most used, and most limited, screen modes available in QB64. In SCREEN 0, there are no graphical commands available, as the screen is a ***text-only*** screen. This means you can **print** to it, but you can not **plot** to it. Commands such as PSET, LINE, CIRCLE, \_PUTIMAGE, and such will \*NOT\* work with SCREEN 0.

 So if there’s no graphics available in SCREEN 0, ***Why is it one of the most used screen modes?***

 Because, as per my simple little Hello World demo back on page 4 illustrates, SCREEN 0 is ***the default screen mode for QB64.*** Unless you specifically specify something else, SCREEN 0 is going to be the default that your programs are written in. Why that is, and what benefits that might offer for you, I’ll be going into below:

### Chapter 03.02.01: Properties of Screen 0

 Screen Width: 80 characters

 Screen Height: 25 lines, and then 50 lines (to be explained below)

 Screen Colors: 16 foreground, 8 background, with a blink toggle (by default)

 Screen Memory Storage: 2 bytes per character. (character & color)

 As you can see from the above, Screen 0 isn’t measured in pixel-size as with the other screens. In fact, Screen 0 ***has an indeterminate pixel size*** by itself, as its total size is completely dependent upon the font used with it. The default font is 8 pixels wide and 16 pixels high, giving a default screen resolution of (8 pixels wide \* 80 characters = 640 pixels, 16 pixels high \* 25 lines = 400 pixels) 640x400 pixels. Note however, that if the font size grows bigger, the whole screen will grow larger, as the overall size is determined by the number of characters and lines of characters that will fit onto the screen.

 ***Overall screen size is going to be determined by (Width = number of characters \* fontwidth), and (Height = number of rows \* fontheight).*** If everything is at the default settings, the screen will be 640x400 pixels in size. If a different font is loaded, or your program changes the width/height in some manner, then the overall size will grow or reduce in relation to the changes made.

###  Chapter 03.02.02: Width of Screen 0

 By default, Screen 0 has a width of 80 characters on a single line. If you want to manually change that value, the way to do so is either to manually specify the desired size of the screen with a \_NEWIMAGE statement, or to use the WIDTH statement, as illustrated in the following examples below:

 SCREEN \_NEWIMAGE(50, 10, 0)

Now, with the above, we’re creating a new screen and setting it to be the one which our program uses, and with the \_NEWIMAGE command, we’re designing it to be 50 characters wide, 10 lines high, and a screen 0 style screen. \_NEWIMAGE(width, height, mode).

If we wanted a screen which could hold 120 characters and 40 rows, we’d specify it as: SCREEN \_NEWIMAGE(120, 40, 0). It’s honestly just that simple!

Alternatively, we could also use the WIDTH statement to alter the size of the screen, such as so:

WIDTH 50, 10

Now, just like before, we’re going to change our screen to become 50 characters wide and to contain 10 rows of information.

In most situations, there’s no real difference in how the two methods work for us, with one noticeable exception: The \_NEWIMAGE method ***creates a new screen***, while the WIDTH method alters the existing one. If you’re going to be using the newimage method repeatedly, you’re going to have to learn how to use \_FREEIMAGE to free up all those old screens from memory as you finish with them (or else your program will suffer from a memory leak and continue to endlessly eat up resources on your machine with each call to the command). If you’re just going to call it once to set the screen to your desired size, then it’ll never become an issue for you. Just keep in mind – repeated calls to \_NEWIMAGE requires the use of \_FREEIMAGE when you’re done with the image. If you’re never done with the image (such as calling it once to set the size of your work window, and then leaving it that size), then you don’t have to worry about freeing anything.

WIDTH, on the other hand, doesn’t have this issue to worry about, so in some ways it’s the simplest and most elegant way to adjust your screen resolution if needed.

###  Chapter 02.03.03: Height of Screen 0

Now, as mentioned above, the height of a Screen 0 program can be altered using the exact same methods as before: \_NEWIMAGE or WIDTH. The first parameter of each command changes width; the second parameter changes height. The same warnings and notes apply here, as they apply above, so I see no reason to repeat them once again.

One thing that is different with regard to the height of a SCREEN 0 program is that SCREEN 0 has a “dual-height” mode. By default, SCREEN 0 has 25 lines with which one can print to the screen.

\*\****HOWEVER\*\****, rather than tossing errors and crashing programs, Screen 0 will ***\*automatically\**** resize your font and height if you use LOCATE to try and print below the available screen. For an example of this, copy and paste the code below into the IDE and compile and run it:

Screen 0

For y = 1 To 50

 Locate y, 1: Print y

 Sleep

Next

\_Delay 5

System

Now, this little program is just going to start counting incrementally for us, with one value going on each line. In the beginning, everything works as we expect and there’s nothing unusual going on, as illustrated in the screenshot below:



 However, once we try to Locate down to the 26th line and print, our whole screen suddenly changes as in the screenshot below:



 As you can easily see, the screen has now automatically altered itself to allow us up to 50 rows of text to be displayed, but these rows are all with characters that are half as tall as previous. Our font is still the same 8 pixels in width, but it is now only 8 pixels high rather than 16. Overall, the screen stays the same size, but the size of the font we’ve used has changed. We’re printing half as tall, which gives us twice as many rows.

 Screen 0 defaults to 25 lines in height, but if you LOCATE below the screen, it will attempt to automatically resize in the manner I’ve illustrated above, rather than tossing an error or ending your program. Be aware of this behavior as it may not be what you actually intend your program to do, but it’s not a glitch or a bug on your part – this is simply the intended behavior as QB64 tries to autocorrect and not toss an error in this one very specific situation.

 (Honestly, I think we ***should*** just toss an error in this type of situation – like we do for all other screen modes when you locate off the screen – but we don’t.) Just keep it in the back of your mind, what’s going on here, in case it ever comes up in your own programs, or in case you ever want to take advantage of this specific quirk for some unique use-case program.

###  Chapter 03.02.04: Colors in Screen 0

 As mentioned previously, Screen 0 has a limited color palette, with a maximum of 16 distinct colors being able to be displayed on the screen at any given time, with the default values looking as illustrated below:



The colors which make up the default palette are: Black, Blue, Green, Cyan, Red, Magenta, Brown, White, Gray, Light Blue, Light Green, Light Cyan, Light Red, Light Magenta, Yellow, and Bright White. Of these colors, all 16 of them are available for the foreground colors (basically the color of your text itself), while only the first 8 are available for use with the background colors (the back drop colors behind the text).

When setting a color value for the foreground, you can specifty the chosen value PLUS 16, to tell the screen to automatically blink that color to draw attention to it. For example:

COLOR 0 + 16 = Blinking Black (it alternates between the black color and the background color)

COLOR 4 + 16 = Blinking Red (an alteration between red and the background color)

COLOR 15 + 16 = Blinking Bright White.

To keep things simple, most people will simply do the math for themselves and write the values as COLOR 16 (which is the same as 0 + 16 for blinking black), or COLOR 31 (which is 15 + 16 for blinking white). I just wanted to point out the actual math at work here for people, as it may become important for someone in the future who is looking to do direct memory access with SCREEN 0 colors.

So, to summerize, you have 16 foreground colors, a flag to set to turn blinking on or off, and 8 background colors – which all add up to become one byte of information, which is all stored together. (Which I’ll be going into more details with in the next heading.)

###  Chapter 03.02.05: Memory Structure of Screen 0

 Like everything we do on our computers, our screens have to be stored somewhere in memory to keep them and preserve them for us. In QB64, if we want direct access to a screen’s memory, we can use \_MEMIMAGE to point a memblock at it, and then access its data directly with \_MEMGET and \_MEMPUT. Unfortunately, being able to access information directly does nothing for us, if we can’t figure out how to decode and interpret that information.

QRT^$@$FAA!!! 🡨There’s information for you. Now, what’s it mean??

In this case it means nothing except Steve was wanting to illustrate his point to help everyone understand what he was saying. 😉

Fortunately, when it comes to the various screen modes covered in this little reference, we know how to read, write, and decode how all the information is stored for us!!

In Screen 0, all information is stored in 2-byte chunks, with those 2 bytes representing each character, and its corresponding color, on the screen. For instance, if your SCREEN 0 screen is only a 10x10 screen (set with WIDTH 10,10, for instance), then it’s going to take 2 bytes to store each character in memory. In this case, it means your screen is using 2 \* 10 \* 10 bytes of memory, for a grand total of only 200 bytes – not megabytes, gigabytes, or terabytes as we’re used to seeing with modern computers, but just 200 bytes of memory to store the whole screen’s worth of information!

As for those two bytes, what they represent are:

 First byte -- The ASCII code of the character to display on the screen.

 Second byte – The ***combined*** color value for that character on the screen.

For an example, let’s take a look at the following code. Feel free to copy and paste this into the IDE and compile it yourself if you want:

Screen 0 'just to highlight that we're working in screen 0

Dim m As \_MEM 'define a mem block

m = \_MemImage(0) 'and point it at our screen in memory

\_MemPut m, m.OFFSET, 65 As \_UNSIGNED \_BYTE 'ASC value of "A" into memory.

Sleep

\_MemPut m, m.OFFSET + 1, 132 As \_UNSIGNED \_BYTE 'combined color code into memory.

Sleep

 Now, if you run the above, it doesn’t actually do anything too impressive, but let me explain what it’s doing here, and why it’s doing it.

First, we make certain we’re in Screen 0. (First line of code.)

Then we create a mem variable and point it at the screen in memory. (Second and third lines of code.)

The fourth line of code then puts a value of 65 into the first byte of memory, If you guys know your ASCII codes, you’ll recognize 65 as being the value for “A”, which explains why there’s an A up in the upper left corner of the screen at this point in our program.

The fifth line is simply nothing more than a SLEEP statement, to give the user time to admire that plain “A” before we go and change it.

The sixth line is where we put a value of 132 into the second byte of memory. (Notice how I’ve moved the pointer over from m.Offset – which is where our screen begins to exist in memory – to m.Offset ***+ 1***?) This second byte of memory controls the color of the text we just placed onto the screen, and, in this case, we just set it to become RED + BLINKING.

***In Screen 0, image data is stored in chunks of two bytes, with those two bytes representing the ASCII character on the screen, and the combined color value of that spot on the screen.***

Understanding the first byte is really very simple: The value is directly equal to the ASCII value of the character itself. 65 = A. 66 = B. 67 = C. 32 = “ “ (space). One byte holds one ASCII value for what character to display onto the screen.

The second byte takes a little bit of decoding to understand fully,as it’s a composite of 3 different pieces of information:

Second byte Mod 16 = Foreground Color

Second byte \ 32 = Background Color

Second byte \ 128 = Blink Toggle

Sound confusing??

 It’s honestly not! Let me break it down for you with a quick example: Let’s say the second byte stored in memory has a composite value of 195. From that we now know that:

 Foreground Color = 195 Mod 16. This comes out to be a value of 3.

 Background Color = 195 \ 32. This comes out to be a value of 4.

 Blink Toggle = 195 \ 128. This comes out to be a value of 1. (1 says “YES, it’s blinking.” 0 says, “No, it’s not.”

So this composite value of 195 is the ***exact*** same as what we’d get with a simple color statement like the following: COLOR 3 + 16, 4 – It’s blinking Cyan on a Red background!

Combined with the value of 65 from before, we now know what the very first character on the screen looks like – it’s a blinking cyan letter “A”, with a red background!

Two bytes of information for each and every character on the screen – and that’s what makes up our screen in memory!

***Side Note:*** If the command \_BLINK OFF is used to disable blinking in Screen 0, then the background has access to the same 16 colors as the foreground. The memory structure stays the same with 2 bytes being used to store each character’s information – one for ASCII character, the other for color, but the composite color value is simplified a bit. With blink off, that second byte represents:

Second byte Mod 16 = Foreground Color

Second byte \ 16 = Background Color

With blinking off, that same value of 195 stored in memory would represent a foreground color of 3, and a background color of 11. Essentially it’d be the same as a COLOR 3, 11 style statement.

###  Chapter 03.02.06: Drawbacks of Screen 0

Now that I’ve covered the basics of what Screen 0 is, let me take a moment to mention the obvious drawbacks of the screen mode – lack of graphics!

Screen 0 is a text-only screen, and as such, there’s absolutely no such thing as graphical commands for the screen mode! No LINE statements. No CIRCLE command. No PSET or \_PUTIMAGE. Screen 0 is purely a text screen, and, as such, it’s naturally limited to what you can do with it.

Of course, just because there’s no in-built graphics commands, that doesn’t mean that you can’t put art into your projects. There’s a whole artform dedicated to what’s known as ASCII Art, such as this little laptop:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|| ||

|| ||

|| ||

|| ||

||\_\_\_\_\_\_\_\_\_\_\_\_||

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

 \\############\\

 \\############\\

 \ \_\_\_\_ \

 \\_\_\_\_\_\\_\_\_\\_\_\_\_\

 Even without the use of our built in graphical commands, one can still add art to their work if they want in Screen 0!

###  Chapter 03.02.07: Advantages of Screen 0

 Since one can’t use any graphics with Screen 0, a lot of folks end up asking, “Why the heck would I want to use Screen 0 for ***anything***, since it’s got such limitations? 16 colors and no graphics? What’s it got going for it that other screen modes don’t have?”

 The answer to that is: “A small blueprint in memory, and ***perfect*** character recognition.”

 Let’s compare two screens, and their memory requirements, for just a moment. One is a Screen 0 text screen that is 80 characters by 25 lines in size. The other is a 32-bit graphical screen which can hold the exact same amount of characters with the default font – 80 characters per line, with a 25 line limit.

 The Screen 0 screen is going to use up 2 \* 80 \* 25 bytes of memory to store all its necessary data. That’s a total of 4,160 bytes of memory being used. The 32-bit graphical screen is going to be 640 pixels wide and 400 pixels tall, and each pixel has a 4-byte color value associated with it. That’s a total of 4 \* 640 \* 400, or 1,024,000 bytes of memory.

 That’s over 200 times more memory required to store the same information on a 32-bit graphical screen!!

 Hows that for a simple reason as to why to use Screen 0, when it’s possible to do so for your needs?

 And, if you guys look up (or have a good memory), you’ll notice that I also mentioned something about a ***perfect character recognition…***

 What the heck was I talking about there??

 That’s when you’re trying to get back information from the screen that somehow magically appeared there. Take a screenshot of this page. Save it as an image. Now, how the heck do you get the information from that screenshot and read it into your work as letters and words, rather than as a picture??

 Hint – something called OCR (Object Character Recognition) has to be worked on that image to turn your graphical screenshot back into a text document.

 There’s only one problem with OCR Technology – it’s impossible to distingish ***every*** character perfectly apart. How many times have you scanned in a document and had your 1’s end up being read as l’s? How many times does an 8 get misinterpeted as a B? Or a O as a 0? Even more impossible: distinish between a printed CHR$(32) and a printed CHR$(255). Both are printed as blank spaces, even if each has a different use in languages. (One is usually a plain space, while the other is a non-breaking space to prevent line breaks between what it joins.) There’s no OCR in the world that’s going to be able to tell the difference between those two spaces!

 ***BUT SCREEN 0 CAN!!***

 Remember how our data is stored in memory with screen 0? It’s not a pixel by pixel storage of color information – it’s data stored in two byte chunks which represent character and combined color value. Our screen would store (32) in the first byte for a normal space, and (255) in the first byte of the next character for the non-breaking space.

 Screen 0 stores character information perfectly for us, so we can always retrieve that information perfectly in the future if we need it, ***and*** it has a much, much smaller footprint in memory than our other graphical screens do. We can’t do graphics on Screen 0, but there’s still plenty of reasons to make use of Screen 0 when you don’t need graphics for your programs.

## Chapter 03.03: Screen Mode 256

 The first thing to note about Screen 256 is that there is ***NO*** actual SCREEN 256. Numeric values that are valid after the Screen statement are all in the range of 0 (zero) to 13, and those are the legacy screen modes available from the days of QBASIC and QB45, which are still supported for older programs.

 Newer programs written in QB64 make use of the \_NEWIMAGE command to create an image, and then we can set it to become our active screen, as follows:

MainDisplay = \_NEWIMAGE(640, 400, 256)

SCREEN MainDisplay

 With the above, we’re calling upon the \_NEWIMAGE command to create a new image for us, and we’re making that image 640 pixels wide, 400 pixels high, and we’re setting it to work in 256-color mode. Then, after the image is created, we’re assigning it to a handle called MainDisplay so we can interact with it as needed in the future in our program. (And, in fact, we’re interacting with it in the immediate future with the second line which then sets that image to become our active screen that we’re now displaying, printing and drawing to, and getting input back from.)

 “SCREEN 256” is actually nothing more than a shorthand way of saying that we’re working with a SCREEN \_NEWIMAGE(width, height, ***256***). There is ***NO*** standard “Screen 256”, so there’s no way that I can tell you guys – or anyone really – what the default width and height of the screen will be. In the end, it’s up to the user to manually specify what they want for the width and height of their screen.

 The only real things which I can help explain here for my dear readers are the color system, memory structure, advantages and disadvantages to using such a screen mode.

### Chapter 03.03.01: Colors in Screen 256

 When it comes to “Screen 256”, the most defining trait of the screen mode is the simple fact that… dum dum de dum (drum roll)… it has a palette of 256 colors available to use with it, and it’s a graphic screen! This color palette looks like the colors on the next page (the image was too large to fit nicely on this page).

 If you look at the image on the next page, you’ll notice that the whole bottom right corner is the same shade of black. This isn’t a glitch of any sort with the screen capture – the fact is: ***ALL THOSE COLORS ARE, IN FACT, THE EXACT SAME COLOR!!***

 Screen 256, really doesn’t have 256 preset colors for use. The actual number of colors is close to 247, and all the colors from 248 to 255 were reserved so the user could define any sort of unique shades or colors which they might want for their program themselves. Color 0 is black. Color 248 is black. Color 249 is black. As is 250, 251, 252… all the way to 255.

Feel free to use the \_PALETTECOLOR command to set these colors to whatever value you wish. (I’ll be explaining and giving examples of the usage of the command later in this reference manual, once I get deeper into that topic, so for now you’ll just have to remember that the colors above 247 are all basically reserved for use with it so you can set and create your own shades and colors for your program.)

***256 Color Palette:***



### Chapter 03.03.02: Memory Structure of Screen 256

 One of the simplest things to explain about Screen 256 is its inherent memory structure. 😊

 ***In Screen 256, each pixel on the screen is represented by a single byte in memory.*** It’s that simple! 256 color images don’t have any sort of foreground or background colors, and they don’t store any character information in them like Screen 0 does. For a 256 color screen, each pixel is a single color, and that single color is stored as an \_Unsigned \_Byte in memory. Color 12 is stored with the byte value of 12. Color 57 is stored with the byte value of 57.

 Honestly, I don’t know what else to say about Screen 256 memory, as the structure is just that simple.

### Chapter 03.03.03: Advantages of Screen 256

 Just as Screen 0 has some distinct advantages over screen modes, so does a 256-color screen. For starters, it’s got the simplest data/memory struture of ***any*** screen available to us. One byte in memory corresponds and represents perfectly one pixel on the screen.

Some of the legacy screen modes use less overall memory for graphical screens – but those modes resort to ***bit packing*** for the process. Accessing their data and reading or writing to it directly is a whole lot more complicated than it is to work with a structure where one byte = one pixel. Think of those packed modes as being the difference between compressing a text file and having to work with it, compared to working with an umcompressed text file. The first, you have to unpack and decode, then read and write to it – the second you just write to it. Now, which of those is simpler and easier to work with??

The second advantage to 256-color screens is the amount of memory they use, and how easy they are to process on your PC. 32-bit color screens require 4 bytes for each pixel on the screen for alpha, red, green, blue channels, and as such they use quite a bit more resources to do what they do. Take a 640x400 screen for example: In 256-color mode, that screen uses 640 \* 400, or 256,000 bytes of memory. The 32-bit screen, however, uses 4 times that amount for a total of 1,024,000 bytes of memory. By itself, that doesn’t seem like it’d be such a large increase, but then consoder the amount of time it’d take to process those two chunks of memory over and over, repeatedly, in a loop running at X frames per second. Now imagine you’re coding your graphics in layers (background layer, player layer, sky layer, ect), so you have multiple screens to render repeatedly. Add in rotations, scaling, and blending…

Now, isn’t it rather easy to imagine which might process faster and using less CPU/GPU resources between the screens which use 256kb of memory verses 1mb of memory. (And remember too, this amount of memory usage is only for a 640x400 scized screen. Many modern systems run 1980x1024, which would scale each screen’s respective memory usage upwards by about 8 times!)

So, in summery:

256-color screens have the simplest memory stucture and ease of direct access to read and write.

256-color screens are uncompressed and have no bit packing involved with them, allowing for quick processing.

256-color screens are ***always*** going to use 1/4th the memory of a 32-bit screen, so if the limited palette is enough for your needs, then they’re going to process and run faster and smoother than 32-bit screens.

### Chapter 03.03.04: Disadvantages of Screen 256

As of the time of this writing (02/02/2022), there’s honestly only two real drawbacks to using 256-color screens.

1. The inherent limition of only having 256 colors to choose from and use.
2. And this one is a real deal breaker for a lot of people with using 256 color screens – ***There’s currently no native support for using \_LOADIMAGE to load 256-color screens from your PC.***

Yep. Sadly, you read that right. QB64, as of the current version, doesn’t offer support for loading 256-color images. Instead, oddly enough, it loads them and then automatically converts them to 32-bit images – which is great if you’re using 32-bit graphics and 32-bit screens, but no so much so for people who want to use them as actual 256 color images.

If you want to load 256 color images into your programs, QB64 will load them and automatically convert them to 32-bit images, which you’ll then have to manually read and write back over to a 256 color screen to convert them back. Currently, the whole process is a pain in the ass, and as such, almost nobody actually uses 256 color images or 256 color screens.

Of course, since no one is using the 256 color images, there’s no real push to add 256 color support to \_LOADIMAGE by the developers. Why waste time and effort to expand a feature that no one’s using, or asking to be able to use? Of course, why would anyone want to use a feature which isn’t fully suported? Especially when the work-around is as simple as just swapping and using 32-bit color mode instead?

### Chapter 03.03.05: Final Thoughts on Screen 256

 Honestly, I find it a little disappointing and sad that 256 color screens don’t get very much usage. (Not even from myself, truthfully.) They have a lot of inherent simplicity associated with them – ease of direct memory manipulation, a small footprint in memory, and a quick and efficient processing time. The biggest problem with them comes heavily from the fact that QB64 simply doesn’t allow us to currently load saved 256-color images and make use of them. If your 256-color resource images are going to automatically convert to 32-bit color images upon loading them, why not just run your program in 32-bit color mode to begin with??

 As long as all your resources are internal (LINE statements, or CIRCLE statements, or PSET, or whatnot), 256-color mode is a wonderfully easy screen mode to work with. I’d recommend it, if possible, for people to use whenever they can.

 The real dealbreaker, however, is just that massive limitation on not being able to easily load 256-color images back into our programs. That’s a hard limit to overcome, and just not worth the hassle in many cases, leaving 32-bit screens as a better and easier choice for most people.

## Chapter 03.04: Screen Mode 32

 And here, we finally come to what could be considered the “bread and butter” of modern programming screens. 32-bit graphical screens, with access to the whole spectrum of 4,294,967,296 colors! ***Screen 32*** is the gold standard by which most modern programs and apps are judged by, and it’s what the vast majority of any graphical programs are written in and using.

 Like “Screen 256”, there truly isn’t any such thing as a “Screen 32”, as the actual syntax is created via the use of \_NEWIMAGE(x, y, ***32***) to set a 32-bit color mode, just as a 256-color screen mode is set using \_NEWIMAGE(x, y, 256). Also, like a 256-color screen, there are no set values for a 32-bit screen, as the user will have to define their custom screen size with the \_NEWIMAGE command.

### Chapter 03.04.01: Colors in Screen 32

 When it comes to colors in a Screen 32, things are a bit more complicated than they are in a 256-color screen. For 256-color screens, each pixel is represented by a value from 0 to 255 which corresponds to the available color palette. For 32-bit color screens, each pixel is represented by a 4-byte value which corresponds to the Alpha, Red, Green, and Blue levels of that pixel, with those 4 values blending together as illustrated below to determine the actual shade of the image.

 If you look to the picture to your right, you’ll see that we have 3 primary colors – Red, Green, and Blue, going clockwise. When Red and Green mix, we get a base shade of Yellow. Green and Blue mix to create Cyan; and Blue and Red mix to create Magenta. When all the colors mix together (as shown by the central overlap), we end up with a shade of White. An absence of all the colors creates Black.

 This blending is the basic premise behind how colors work in a 32-bit screen, with the final byte containing alpha values which dictate how transparent a color will be and how strongly it’ll blend into the existing screen.

 \****ALL\**** the pixels on a 32-bit screen are represented by ***UNSIGNED LONG*** values. Let me emphasize that once again: 32-bit color values are ***UNSIGNED LONG*** values! Use of incorrect and insufficient variable types can and will end up causing you issues in your programs. For a quick example of how various variable types contain different values, let’s compile and run the following little program:

|  |
| --- |
| Dim BlueLong As Long, BlueSingle As Single, Blue\_UnsignedLong As \_Unsigned Long |
| BlueLong = \_RGB32(0, 0, 255) ' All blue for a color |
| BlueSingle = \_RGB32(0, 0, 255) ' All blue for a colo |
| Blue\_UnsignedLong = \_RGB32(0, 0, 255) ' All blue for a color |
|  |
| Print Using "###############"; Blue\_UnsignedLong |
| Print Using "###############"; BlueLong |
| Print Using "###############"; BlueSingle |

 If you compile and run the above, you’ll see that the value for 32-bit BLUE varies by quite a bit, depending on the type of variable used to store it. As an integer, the value is -16776961. As a single, the value is 4278190336. As an unsigned long, the value is 4278190335.

 Now, first of all, let me state and emphasize: ***THE SINGLE VALUE OF 4278190336 is WRONG!!*** Absolutely, unarguably, the value is wrong! The value that 4278190336 represents for us is a value with ZERO alpha. It’s completely transparent!

 Now, we sure as hell didn’t ask \_RGB32 to return us a ***transparent*** blue, now did we?? Nope! We just asked for the value of a deep blue. So why then, does the single value return a number that is transparent?

 ***Because we exceeded the limits of our variable type!***

 Let’s take, for example, the value of 1E20. Converted into a non-scientific notation, it’d hold a value of 100,000,000,000,000,000,000.

Now, add 100 to that value: 100,000,000,000,000,000,100.

Now convert that value back to scientific notation: 1E20.

Yep! Due to rounding to the nearest exponent, there’s no difference between 100,000,000,000,000,000,000 and 100,000,000,000,000,000,100 once they get converted to scientific notation. The system only keeps track of the value to the nearest exponent.

And, if we actually just PRINT BlueSingle, without using the PRINT USING format, we see that’s exactly what’s happening to us here. BlueSingle = 4.27819E+09. We’re to the point where we’re losing 5 digits worth of values to rounding. The values after the decimal preserves 4 digits worth of information, but we’re looking at a E+09 for a total of 9 digits after that 4.

Single variable types simply can not preserve all the values that we have in 32-bit colors – not without losing a lot of shades due to scientific rounding and the inherent limitations of the variable type.

***You should never, NEVER, NEVER-EVER, use SINGLE variables to hold 32-bit color values.*** If you do, you can expect issues with many of your colors to lose their blue values due to rounding limitations. You’ve been warned here first! 😉

### Chapter 03.04.02: Memory Structure of Screen 32

 When it comes to a 32-bit screen, all of our color information is stored pixel-by-pixel, just the same as a 256 color screen – with one difference: It’s stored in four bytes of information, instead of just one. For a 256-color screen, our color values can only range from 0 to 255, so a single byte perfectly holds that information. For a 32-bit screen, our colors fall into a range which is composed of four different values – Alpha, Red, Green, and Blue. (Not so subtle hint: That’s why we used the \_RGBA and \_RGBA32 commands to get 32-bit color values for us.)

 Alpha represents transparency of an object, and it’s in a range from 0 (completely transparent) to 255 (completely solid), so an alpha level of 128 would represent a 50% faded/blended color.

Red, is of course the red value of our color from 0 to 255, just as Green and Blue are the same respectively. When it comes to blending colors, you can always trial and error values to get a color that you like, or you can always use one of the many color pickers which are found on the web. I’m not going to go into any tutorial here over what values make what colors, but I’ll probably cover that topic elsewhere when I get down to the actual COLOR command itself.

As far as any 32-bit image is concerned in memory, it’s generally stored pixel-by-pixel, in an Alpha, Red, Green, Blue format. You can easily calculate the size of your uncompressed image in memory with just: Memory\_Usage = \_Width(image) \* \_Height(image) \* 4. So, a 100 by 100, 32-bit image, would use 40,000 bytes of memory.

In memory, the pixels are stored sequentially in an Alpha, Red, Green, Blue format. The first four bytes of screen memory are going to be the first pixel on the screen, the next four bytes are going to be the second pixel on the screen, down to the last four bytes being the last pixel on the screen. Simple enough, right?

An easy set of formulas which you can use to interact with 32-bit color values in memory is:

Memory\_Position = (Y \* \_Width(image) + X) \* 4

For example, to find the memory position of the pixel at Point(0,0), it’d be (0 \* \_Width(image) + 0) \* 4, which would be at position 0, which should make sense to everyone as it’s the first pixel on the screen!

To find the memory position of the pixel at Point(10,10), on a 100x100 image/screen, it’d be at (10 \* 100 + 10) \*4, which would be at position 4040.

That’s honestly all there is to it! Kolor~& = \_MEMGET(m, m.offset + 4040, \_Unsigned long) would get that color value from memory, and \_MEMPUT m, m.offset + 4040, Kolor~& would put whatever value you’d like into Point(10,10). Nothing complicated at all about that, now is it?

### Chapter 03.04.03: Advantages of Screen 32

 As I’ve mentioned elsewhere, the main advantage to using 32-bit color screens is simply the fact that they tend to be the standard for just about anything graphical anymore, and that they can represent and make use of the whole color spectrum that our machines allow. For the absolute best photo-quality images, you’re going to want to use 32-bit color images and screens. There’s just no getting around that.

### Chapter 03.04.04: Disadvantages of Screen 32

 As also mentioned previously, when discussing 256 color screens, the only real drawback to 32-bit screens is the simple fact that they take up and use four times the memory of a 256 color screen. This increased memory usage, and the way that colors blend and fade together with various alpha levels, can lead to 32-bit images and screens processing slower and causing more issues of lag in programs than 256-color images do.

 Of course, this problem of more memory being used and slower performance isn’t honestly something a programmer has to worry about too often on modern systems. Take a large 1920 x 1080 image (1080p HDTV resolution), for example – that’s only 9,294,200 bytes of memory used. 8 MB of memory… compared to even some of the weakest of modern computers which come with 2 GB of video memory!

 Unless you’re dealing with old systems, or are using literally hundreds or thousands of large images, you shouldn’t see a lot of issues with using 32-bit screens. Just keep in mind, however, that there’s a lot of folks who still consider QB64 to be nothing more than “just another BASIC like QB64”, and they want to run it on whatever old, dust-covered dinosaur of a PC that they managed to dig up out of dustballs in their garage. While your code might run flawlessly on 32-bit screens with any modernish computer (like from the last 10 years or so), you can probably expect it to have issues on someone’s old Pentium II machine.

If you’re trying to code for ancient hardward, then modern screens probably aren’t what you’re going to want to be coding in! (Heck, I don’t even think you’d be able to get an old CGA/EGA/VGA video card to even run in “true color” mode. Back in the days of some of those original dinosaurs, you were lucky to have any color on them besides a few shades of green!)

32-bit images are the modern format. They work fine on modern machines. Caveman equiptment… not so much – and that’s the only real disadvantage to 32-bit screens!

### Chapter 03.04.05: Final Thoughts on Screen 32

Personally, I’ve always liked working with 256 color screens for myself. Most of what I do isn’t “photo quality”, and I’m perfectly happy to stick with a smaller color palette for the smaller footprint in memory. I don’t usually need to fade one image onto another, and most of the time, I doubt if I ever use more than 20 or 30 defined colors in a program. I’m not a graphical artist. I don’t tend to worry about subtle levels of shading and such. I’m a simple guy who tends to write simple programs. 256 color screens have always been my favorite…

At least, that \*used\* to hold true. Since QB64 doesn’t (currently) support loading 256 color images via \_LOADIMAGE (it auto-converts them up to 32-bit images instead), I’ve been using 32-bit images almost exclusively for all my work. From what I’ve learned, after using 32-bit images for a while, is that ***I’ll probably never go back to 256 color images***!

The reason?

I find it much simpler to keep up with color values via their HEX codes than I do via some 256-color index. For example, in a 256-color screen, I know that COLOR 40 is the color code for DEEP RED. (I know this one from years of personal experience.)

In a 32-bit screen? That value is &HFFFF0000.

Now, does that seem confusing at first? It’s really not, once you break that hex value down into it’s 4 components: Alpha, Red, Green, Blue!

&HFFFF0000, breaks down as:

&H – this is QB64 code saying, “A hex value is coming next!”

FF – those first two characters are hex for the alpha level of our color, or in this case, 255.

FF – the next two characters are hex for the red level of our color, which is again 255.

00 – two characters to represent our green level. (0 in this case.)

00 – the last two characters represent the blue level. (0 in this case.)

&HFFFF0000 – is hex for full alpha, full red, no green, no blue.

How’s that for simple?!!

&HFF0000FF – this would be full alpha (a solid color), with no red, no green, and full blue.

&HFF00FF00 – this is full alpha (solid), with no red, full green, and no blue.

&HFFFFFF00 – and this one? Full alpha, full red and full green… Red and Green make YELLOW!!

Now, quickly, what’s the color code for Green, Blue, or Yellow in a 256-color screen?? Are you ***certain*** the value you’re thinking is correct? What if your palette got remapped to allow for colors which aren’t on the standard QB64 256-color palette?

Since I don’t program on dinosaurs like I used to twenty years ago, I find myself no longer needing to bother with limiting myself to just 256 color palettes. I don’t have to have a chart printed out with all the colors and numbers beside it so that I can find the one I’m looking for, anymore. Nowadays, I tend to stick to 32-bit screens, and tend to keep all my colors straight either via \_RGBA or via using HEX values for my colors.

\_RGBA(255, 0, 0, 255) – full red, full alpha, nothing else… Everyone should be able to understand what this value represents. \_RGBA(128, 0, 0, 255) – 50% red, full alpha, nothing else… Is it a stretch to think that everyone should be able to figure out that this color is still red – just not as dark of a shade of red as the previous one?

And yet, who knows what COLOR 174 of a 256-color screen is, without having a palette guide handy to look at??

32-bit colors are simple to remember. Easy to use. They offer full levels of shading, mixing, blending, and all your possible display values.

Unless you’re just planning to code for archaic systems, or for those with no real video memory to speak of, 32-bit screens are the best way to go. There’s a reason why they’ve become the modern standard, after all.

# Chapter 04: Optimizing Programs

 One thing that people often ask on the forums and in the various Discord chat channels and such is always, “What’s the best way to optimize a QB64 program so it’ll run faster?” Let’s take a moment and look at a few surefire tips which always helps to improve program execution speed and performance.

## Chapter 04.01: DO over FOR loops

Here’s one simple, irrefutable law for QB64 programs – DO… LOOP will ***always*** be faster than a FOR… LOOP. If you’re trying to optimize your program for speed, then swap out the FOR loops for DO loops, and it’ll always run a little faster than before.

 Why?

 To answer that, let’s take a look at two different loops and how they translate into the C-source which ends up getting compiled into our EXE for us by gcc.

Example #1: A basic FOR.. LOOP from 1 to 10

|  |
| --- |
| FOR I = 1 TO 10 |
|  ‘stuff |
| NEXT |

Example #2: A basic DO.. LOOP from 1 to 10

|  |
| --- |
| I = 1 |
| DO |
|  ‘stuff |
| I = I + 1 |
| LOOP UNTIL I > 10 |

Type the first little program into QB64, and then look inside the internal folder and find main.txt. main.txt is going to have the vast majority of our translated code located in it, and in this case, that’s where we’ll find the relevant translations for these snippets. Both end up translating to the following:

Example 1, Translated into C-code:

|  |
| --- |
| S\_1:; |
| fornext\_value2= 1 ; |
| fornext\_finalvalue2= 10 ; |
| fornext\_step2= 1 ; |
| if (fornext\_step2<0) fornext\_step\_negative2=1; else fornext\_step\_negative2=0; |
| if (new\_error) goto fornext\_error2; |
| goto fornext\_entrylabel2; |
| while(1){ |
| fornext\_value2=fornext\_step2+(\*\_\_SINGLE\_I); |
| fornext\_entrylabel2: |
| \*\_\_SINGLE\_I=fornext\_value2; |
| if (fornext\_step\_negative2){ |
| if (fornext\_value2<fornext\_finalvalue2) break; |
| }else{ |
| if (fornext\_value2>fornext\_finalvalue2) break; |
| } |
| fornext\_error2:; |
| if(qbevent){evnt(1);if(r)goto S\_1;} |
| fornext\_continue\_1:; |
| } |
| fornext\_exit\_1:; |

Example2, Translated into C-code:

|  |
| --- |
| \*\_\_SINGLE\_I= 1 ; |
| do{ |
| \*\_\_SINGLE\_I=\*\_\_SINGLE\_I+ 1 ; |
| dl\_continue\_1:; |
| }while((!(-(\*\_\_SINGLE\_I> 10 )))&&(!new\_error)); |
| dl\_exit\_1:; |

Now, just looking at those two sets of code, it’s easy to tell – even without knowing a single bit of c-programming, that the second one is much shorter and cleaner than the first. WHY??

 Take a look at the first few lines of the FOR translated code, for a hint:

|  |
| --- |
| fornext\_value2= 1 ; |
| fornext\_finalvalue2= 10 ; |
| fornext\_step2= 1 ; |

 FOR I = 1 to 10 ***STEP 1*** – You may not have written the STEP 1 in your code, by it’s there by default. With a FOR… LOOP, the third segment of the FOR statement is for the value that we increment by. Also remember that we \**could*\* increment by a *negative* number (FOR I = 10 TO 1 STEP -1), so the translation has to account for that as well, which is what we see in the lines here:

|  |
| --- |
| if (fornext\_step\_negative2){ |
| if (fornext\_value2<fornext\_finalvalue2) break; |

 So FOR…NEXT loops have to account for all the possible ways we might end up writing them. Count forward. Count backwards. Count by fractional amounts.

 Those DO… LOOPs on the other hand? They’re pretty much a straight translation, line for line, for what we write in QB64. It’s up to the user to specify the starting point, the ending point, and the incremental value. The translation doesn’t need to sort out if it’s supposed to count forwards, or backwards, or anything else. All of that is left for the user to write in their loops themselves, and it ends up with much cleaner and shorter code. Less IF conditions to check (IF we’re counting up or counting down… If we’re greater than the end result, or less than the end result…), equals less time the computer has to spend inside that loop.

 ***DO…LOOP will \*always\* be at least a little faster than an equivalent FOR…NEXT loop.***

## Chapter 04.02: Remove MATH from loops!!

One of the most important ways to improve program execution speed is simply by removing math from outside of a loop. The basic concept here is really rather simple – the less instructions we give the program to run inside the loop, the faster the program can repeat that loop – yet it’s something which people often overlook for convenience’s sake. One of the places where I see this type of issue the most is with dealing with \_MEM, which is a little complex to begin with for many people. Often, we’ll see a batch of QB64 code which will look like the following:

|  |
| --- |
| FOR x = 0 to \_WIDTH - 1 |
|  FOR y = 0 to \_HEIGHT – 1 |
|  \_MEMPUT m, m.offset + y \* \_WIDTH \* 4 + x \* 4, &HFFFF0000 AS \_UNSIGNED LONG |
|  NEXT y |
| NEXT x |

Now, the above here is basically just a simple means to use \_MEMPUT to turn the whole screen Red. To explain it a bit, let me break down that \_MEMPUT statement:

\_MEMPUT m 🡨 the m here is basically just the memblock which we defined to point to our screen.

m.offset 🡨 where that screen starts in memory

y \* \_WIDTH \* 4 + x \* 4 🡨 this gives us the position of a single pixel in memory (Think PSET (x,y) in mem.)

&HFFFF0000 🡨 This is just the 32-bit color value for Red.

Now, looking at the math above, that’s quite a few calculations that we’re having to do over and over and over inside that loop. Is there *any* way we can factor that down and reduce the number of steps we have to take to get that value??

First, we have a common multiplier of 4. We can factor it out.

(y \* \_WIDTH + x) \* 4 🡨 Yay! One less calculation to do!

But, looking at it like this, can’t we just factor that 4 out completely and process it inside the FOR…LOOP itself?

|  |
| --- |
| FOR x = 0 \* 4 to (\_WIDTH – 1) \* 4 |
|  FOR y = 0 \* 4 to (\_HEIGHT – 1) \* 4 |
|  \_MEMPUT m, m.offset + y \* \_WIDTH + x, &HFFFF0000 AS \_UNSIGNED LONG |
|  NEXT y |
| NEXT x |

Well that just completely removed the \*4 from out of the innermost loop. Is there anything else we could factor out from that math, and process once and be done with it? How about the \* \_WIDTH?

Sure we can!

|  |
| --- |
| FOR x = 0 to (\_WIDTH – 1) \* 4 |
|  FOR y = 0 to (\_HEIGHT – 1) \* 4 \* \_WIDTH |
|  \_MEMPUT m, m.offset + y + x, &HFFFF0000 AS \_UNSIGNED LONG |
|  NEXT y |
| NEXT x |

Well, that looks a little less complicated to calculate than it did before, now doesn’t it? Just a few basic math calculations, with no need to multiply, look up \_WIDTH values, or do anything extra inside the loop. 2 addition operations verses 2 addition, 3 multiplication, and 1 function lookup… Anyone want to care to take a guess which is going to be faster? (And usually quite a bit faster once it repeats a few million times each time the routine is called.)

***Move your math operations outside of the loop as much as possible, so that you aren’t having to calculate them inside the loop.***

## Chapter 04.03: Code Layout can play a large role in optimization!

 And here’s one which might be a little counter-intuitive, as it sometimes ends up creating ***more*** code inside your project, but \****how***\* your code is laid out can truly affect performance speeds. For an example of this, let’s take a look at two very simple little example snippets:

Example #1:

|  |
| --- |
| FOR i = 1 TO 1000 |
|  SELECT CASE userInput$ |
|  CASE “A” TO “Z” : ‘do stuff with uppercase input |
|  CASE “a” TO “z” : ‘do stuff with lowercase input  |
|  END SELECT |
| NEXT |

Example #2:

|  |
| --- |
|  SELECT CASE userInput$ |
|  CASE “A” TO “Z” :  |
|  FOR I = 1 TO 1000 |
|  ‘do stuff with uppercase input |
|  NEXT |
|  CASE “a” TO “z”  |
|  FOR I = 1 TO 1000 |
|  ‘do stuff with lowercase input |
|  NEXT |
|  END SELECT |
| NEXT |

Now, looking at these two sets of code examples, it’s obviously easy to see that the first set of code is fewer lines than the second set of code. A \****lot***\* of times, we’ll see people write code using the first layout simply because it is so much shorter and easier to code. The only thing here is… ***the second set of code is more efficient and will run faster!!***

It’s counterintuitive, I know, but it’s true. Our brains always tries to tell us, “Shorter programs are more efficient than longer programs,” but that unfortunately doesn’t always hold true. Take a moment to ***actually look at the program flow of the two examples!***

 Example 1 is fairly straight forward – Start a loop to do something multiple times. Make a decision inside that loop in case you have uppercase or lowercase input, and do stuff. Repeat loop until finished.

 Example 2 is also fairly straight forward – Make a decision to decide if input is lowercase or uppercase. Start a loop to do something multiple times. Do stuff. Repeat loop until finished.

 Can you see the big change for what we’re processing inside the loop?? ***Make a decision inside that loop in case you have uppercase or lowercase input…*** In the second example, we made that decision to begin with, whereas in the first example we *make that decision over and over each loop*!

 1000 decisions to make (in this example case), verses 1 decision to make. Which do you think is going to be faster – even though it requires more actual code to be written than the other??

 ***Code layout can cause a large change in the performance of a program. Reduce code inside repetitions as much as possible (***which is basically the same as the previous tip above about removing math from a loop***) if you’re trying to maximize speed and performance as much as possible.***