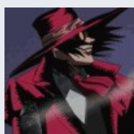
Author | Topic: Altering/Adding to QB64 itself (Read 173 times)

SMcNeill
QB64 Developer



Altering/Adding to QB64 itself

« **on:** August 18, 2019, 09:28:14 PM »

So, after several long days of pounding my head against the keyboard, I've finally managed to sort out the process of how to add an external library to QB64 and get it to work for us, and I thought I'd share with you guys, so perhaps more people would feel comfortable with altering the source and helping to tweak out the bugs in QB64, or add future enhancements to the language for us.

First thing to realize is QB64 is broken down into two distinct parts for us -- the *.BAS side of things, and the *.CPP side of things. When adding to the language, you'll need to make changes to both sides, before you'll be able to take advantage of your work.

On the *.BAS side of things, you can find the QB64 source inside, fittingly enough, the /source folder of QB64. QB64.bas is the program we compile which gives us our IDE and which translates our programs into C-code for us. It's been broken down into several modules, to make navigating around in it easier, so you need an idea of *what* you'd like to change, before rooting around in it.

* For general changes to the language (like a glitch in how a keyword is translated), you'd want to go into QB64.bas itself and make your alterations.

* For general changes to the IDE, and how things appear on the screen, you'd most in likely want to head into source\ide\ide_methods.bas and make changes.

* To register a new command, you'd want to source\subs_functions\subs_functions.bas.

* To register that command so that QB64 colors it properly, you'd want source\ide\ide_global.bas

Anything else, you can just ask one of us who muck around in the internals quite a bit, and we'll try and help you find the proper place to make your changes. Truth is, even I often have to root around for a bit sometimes, before I find the proper placement for whatever I'm wanting to do. There's no shame in asking others to help point you to where you might want to make your changes, and patience is the keyword until you familiarize yourself to the internals quite a bit.

On the C-side of things, generally speaking, all a user will normally have to work with and change is two main files:

* internal\c\libqb.cpp -- the spot where we generally write and store the bulk of our C-code

* internal\c\qbx.cpp -- the spot where we generally need to declare any c variables/routines which we make use of in internal\c\libqb.cpp

---

And with those little guidelines in mind, let me walk you guys through the general process of adding a new command to QB64.

First thing for you to do is grab a fresh copy from the repo, and save it in its own folder. Don't muck up your working copy of QB64 while playing around and trying things out. Once you have a fresh version to work with, place an idea of what you'd like to do inside your head.

Your idea is currently, "***I'd like to add a cheesy little command to add a zero to a number! (Basically multiply it by 10.)***"

To see this idea to completion, you'd first need to open internal\c\libqb.cpp in your favorite text/c editor. Navigate to the end of the file and write a simple set of C-code to do what you want. (If you don't know any C -- I didn't when I first started changing QB64 -- GOOGLE and BING are your friend. Look for working examples and see if you can apply them to your needs, if possible.) Writing a command to simply add a zero to a number isn't that hard:

Code: C++ [Select]

```
1.  double addzero (double user_variable){
2.      double temp;
```

```
3.     temp = user_variable * 10;
4.     return temp;
5. }
```

That's basically it! Save your changes and pat yourself on the back. The hard (for me, at least), part is now finished. You've got the basic c-translation into the source!

Now, go into internal\c\qbx.cpp and declare your procedure. (Remember, C requires all our variables and functions to be declared before we make use of them.) That's as simple as basically just copying the first line of your routine and adding a semi-colon to the end of it, before pasting it into a good spot in qbx.cpp, like so:

Code: C++ [Select]

```
1. double addzero (double user_variable);
```

Now that you've made both changes to the c-code, it's still completely worthless, and for two distinct reasons:

1) Your changes are only in the source code and not in the libqb.o library which we make from that code and link to.

2) You still haven't added your code into the *.BAS side of things yet.

Issue 1 is really simple to fix -- simply run the **internal/c/purge_libqb_only.bat** file. The batch file will delete the old library and QB64 will see it's missing and build a fresh one, taking advantages of the changes you made to the code. (And also glitching out with any typos or missing semi-colons which you might have inadvertently left in your code. Be certain to look closely at any C-compiler failed messages when testing out your BAS code beyond this point, once you start adding it to the language.)

To add your command to the BAS side of things, navigate to source\subs_functions\subs_functions.bas and open that file. (Start from the main QB64 directory; you won't find it buried anywhere amongst the internal c code...)

Navigate to the end of the file, and you'll see where we register our new commands into QB64. Try to think of a similar command to what you're looking to do, scroll up and find it, and copy it and paste it to the end of the file for an easy template of what you'd need to change. In this case, _D2R will be my example code as it's one I've added into the language in the past, and it basically does what we want -- take one number in, spit another number out.

This is what the structure for _D2R looks like:

Code: QB64 [Select]

```
1. clearid
2. id.n = "_D2G"
3. id.subfunc =  1
4. id.callname = "func_deg2grad"
5. id.args = 1
6. id.arg = MKL$(FLOATTYPE - ISPOINTER)
7. id.ret = FLOATTYPE - ISPOINTER
8. regid
```

To break that structure down into its parts, it's basically:
clearid = the way to clear the last command's values so we don't corrupt this ones.
id.n = the name of your command as it'll appear in QB64
id.subfunc = 1 if a FUNCTION, 2 if a SUB
id.callname = the name of your C-command as it appears in libqb.cpp.
id.args = the number of parameters in your command
id.arg = the types of variables which you're expecting to pass to the c-routine
id.ret = the types of variables which you're expecting the c-routine to pass back to you
regid = the command to register that new command internally into QB64

There's other options available, as needed, so it's a good idea to find a template from an existing command very similar to what you want, and use it to go by to make certain you don't leave anything out.

For us, what we'll do is copy that structure from _D2R, scroll down to the end of the file, paste it in, and then change it so that it looks like:

Code: QB64 [Select]

```
1. clearid
2. id.n = "_ADDZERO"
3. id.subfunc =  1
4. id.callname = "addzero"
5. id.args = 1
6. id.arg = MKL$(FLOATTYPE - ISPOINTER)
7. id.ret = FLOATTYPE - ISPOINTER
8. regid
9.
```

Save that code, and you're more or less done! Wipe the sweat from your brow and drink a big gulp of soda! YAAAAY!!

You've written the C-code, declared it, cleared the old c-libraries, and registered your new command into QB64!

And you open QB64.exe, type in your new command, and get an ERROR!!

WHY?!?!

Because, just like how you needed to clear those old c-libraries with purge_libqb_only.bat, so you're not working with the old version of things, you can't work with the old version of QB64. That old EXE in your folder isn't going to have any of your current changes in it, now is it?

Instead, use QB64.EXE to open source\qb64.BAS and compile it. DON'T save to source directory, or else your new EXE will complain "can't find the folder structure I'd expect!!! GAHHHHH!!", and crash. If it does complain and crash, simply move the EXE you just created from the /source folder, to the main QB64 folder and then run it.

At this point, you've now added your command into the language! Test it out!!

Code: QB64 [Select]

```
x = 1.1
PRINT _ADDZERO(x)
```

Compile, run, and see a lovely 11 printed to your screen.

That's it! You've now added your first new command into the language!

EDIT: I forgot to register that new command so that it'll be nice and colored in the IDE! /whoops!

To have your command colorize itself properly, head into source\ide\ide_global.bas and scroll down to around line 120ish or so (line position is always subject to moving as things are added/altered in the language in the future). What you want to look for is something like the following:

Code: QB64 [Select]

```
DIM SHARED listOfKeywords$, listOfCustomKeywords$, customKeywordsLength AS LONG
listOfKeywords$ = "@?@$CHECKING@$CONSOLE@ONLY@$DYNAMIC@$ELSE@$ELSEIF@$END@$ENDIF@$EXEICON
listOfKeywords$ = listOfKeywords$ + "_ERRORLINE@_EXIT@_EXPLICIT@_FILEEXISTS@_FLOAT@_FONT@
listOfKeywords$ = listOfKeywords$ + "_GLCOPYTEXSUBIMAGE2D@_GLCULLFACE@_GLDELETELISTS@_GLD
listOfKeywords$ = listOfKeywords$ + "_GLPOPATTRIB@_GLPOPCLIENTATTRIB@_GLPOPMATRIX@_GLPOPN
listOfKeywords$ = listOfKeywords$ + "_SOFTWARE@_SQUAREPIXELS@_STRETCH@_ALLOWFULLSCREEN@_A
listOfKeywords$ = listOfKeywords$ + "_DEFLATE$@_INFLATE$@$COLOR@"
```

That's the list of keywords the IDE uses to colorize things all pretty for us. At the last line, simply add your command name to the list, with an @ appended to the end of it.

Code: QB64 [Select]

```
listOfKeywords$ = listOfKeywords$ + "_DEFLATE$@_INFLATE$@$COLOR@_ADDZERO@"
```

Save, open QB64.bas, and recompile the source so you have a new version with your changes in it.   From now on, your keyword will show up all nice and pretty and in color in the IDE, just like all the others.

YAAAAY!!!

NOW, I think we're finally done.   :P

« Last Edit: August 19, 2019, 01:00:07 AM »

https://github.com/SteveMcNeill/Steve64 — A github collection of all things Steve!

SMcNeill

QB64 Developer

Re: Altering/Adding to QB64 itself

« **Reply #1 on:** August 18, 2019, 09:28:56 PM »

QB64 Code Etiquette: If you're just coding things for yourself, you can skip this part. If you want to publish into the official version of QB64, kindly follow these rules:

* routines used by QB64, in libqb.cpp, are preceded with **func_** or **sub_**. You'll see this naming convention used throughout libqb, and it helps to quickly jump to a command to work on it. (Search for func_rnd, for example, to quickly find the RND code.)

* note, with this convention, there's TWO underscores for new keywords usually. (func__dest for _DEST, as an example).

* subs used internally by the c-code generally don't follow this pattern. If you see **whatever=123**, you can be pretty certain that variable doesn't show up in QB64 anywhere. (An exception to this guideline is the window handles and such, which we shared with QB64 long after they were originally added. For whatever reason, Galleon assumed nobody would ever need to see them.)

* QB64 command names should be in ALLCAPS, and since it's something new to the language, preceded by an UNDERSCORE.

(Other general etiquette stuff to follow, as my poor befuddled brain reminds me to mention what's now became second nature. Generally though, try to put stuff in the proper places and follow existing conventions when you notice them — and don't get peeved if somebody else makes little alterations to your code to help it fit in better. For example, if addzero was pushed into the repo, it'd probably be changed by one of the regular developers to become func__addzero.)

« Last Edit: August 19, 2019, 01:26:04 AM »

https://github.com/SteveMcNeill/Steve64 — A github collection of all things Steve!

SMcNeill

Re: Altering/Adding to QB64 itself

« **Reply #2 on:** August 18, 2019, 09:29:19 PM »

QB64 Developer

Quote from: SMcNeill on August 18, 2019, 09:28:56 PM

> Post spot reserved for future advanced notes, before others start to comment.

+1

Report to moderator Logged

https://github.com/SteveMcNeill/Steve64 — A github collection of all things Steve!

SMcNeill

QB64 Developer

Re: Altering/Adding to QB64 itself
« **Reply #3 on:** August 18, 2019, 09:31:20 PM »

Quote from: SMcNeill on August 18, 2019, 09:29:19 PM

> +1

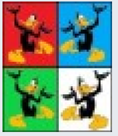
+2

Feel free to post any questions, comments, or specific concerns down below this post. I'll take what info is gathered and edit the two blank posts above (and this one), with any FAQ and new information, as needed, in the future. ;)

Report to moderator Logged

https://github.com/SteveMcNeill/Steve64 — A github collection of all things Steve!

TempodiBasic



Re: Altering/Adding to QB64 itself
« **Reply #4 on:** August 18, 2019, 11:55:26 PM »

Wow Steve
I can see an HowToDo for QB64 expansions....
very useful!
It saves so many times to spend to understand QB64.bas structure and its alter ego qbx.cpp libqb.cpp.

Thanks to share knowledge

Report to moderator Logged

Programming isn't difficult, only it's consuming time and coffee

luke

QB64 Developer

Re: Altering/Adding to QB64 itself
« **Reply #5 on:** August 19, 2019, 02:47:48 PM »

Luke's quick guide to using the internal qbs string type:

Note: qbs_new_* functions do not copy data.
qbs_new_txt*(char *s): Create qbs from C-style string. Temporary, readonly.
qbs_new_txt_len(char *s, int len): Create qbs from raw data of length len. Temporary, readonly.
qbs_new_fixed(void *offset, int len, int tmp): Create qbs from raw data of length len. Fixed, possibly temporary.
qbs_new(int len, int tmp): Create empty string of size len. Possibly temporary.
qbs_maketmp(qbs *str): Make str into a temporary string.
qbs_set(qbs *dest, qbs *src): Replace content of dest with copy of content of src. Return the new string descriptor for dest (may have changed to allocate new memory)

"Temporary" strings are automatically garbage-collected after each QB line by a call to qbs_cleanup().
References to cmem can *mostly* be ignored. cmem is literally a 1MB block that emulates real-mode conventional memory. In rare circumstances strings are marked as being "in_cmem", but in general this flag is false. Best I can tell it's all just to support SADD, VARPTR and VARSEG. So, you know, I probably won't necessarily bite you if your super new feature doesn't play nicely with VARPTR.

« Last Edit: August 19, 2019, 02:58:55 PM »

Report to moderator Logged

Cobalt

At 60 I become highly radioactive!

Re: Altering/Adding to QB64 itself
« **Reply #6 on:** August 19, 2019, 05:10:26 PM »

Awesome Steve, now I can find every thing I've forgotten! Now I just need this kind of information on how to get it pushed in github so I can get the rest of the bit handling routines added to QB64.

Report to moderator Logged

Granted after becoming radioactive I only have a half-life!

SMF 2.0.15 | SMF © 2017, Simple Machines
XHTML RSS WAP2