q\_core Internals

This document explains

* function details from q\_core.lua
* library loading process
* support for dynamic compilation
* changes required to achieve DC
* assumptions
* TODOs

q\_core is a heart of Q. It allows operator developer to access C functions by loading libraries (“.so” files) including libq\_core.so, exposes API to register the C functions, allows us to measure function performance (in terms of time taken) in debug mode etc.

**Location:** Q/UTILS/lua/q\_core.lua

Below keywords are used in the document

Q\_INC\_DIR - $Q\_ROOT/include/

Q\_LIB\_DIR - $Q\_ROOT/lib/

Q\_TMPL\_DIR - $Q\_ROOT/tmpl/

## q\_core keywords/data-structures

**q\_add** - a reserved keyword represents the q\_add() function in q\_core

**function\_lookup** - structure maintaining all the dynamically registered functions

**qc** - a object having all statically defined functions

**libs** - maintain references to dynamically loaded libraries to avoid them to mark for GC

**qt** - qt is essentially an empty table with meta methods like \_\_index() & \_\_newindex()

## q\_core function details:

q\_core has below functions

### load\_libs()

This function performs the library loading activity with the help of header file & “.so” file. It expects a header file path as a argument and finds respective “.so” file, which it loads with the help of ffi.

After loading the library, it adds entry to ‘function\_lookup’ and ‘libs’ structures that q\_core maintains internally.

### add\_libs()

This function helps q\_core to load all “.so” from Q\_LIB\_DIR and respective “.h” files from Q\_INC\_DIR when q\_core is loaded first time.

For each header file in Q\_INC\_DIR dir (except “q\_core.h”), there should be a respective “.so” file in Q\_LIB\_DIR dir

Convention for header file and respective “.so” file should be as below, for a function ‘fn\_name’

header file - **<fn\_name>.h**

“.so” file - **lib<fn\_name>.so**

**“libq\_core.so”** in Q\_LIB\_DIR and **“q\_core.h”** in Q\_INC\_DIR should be present which get loaded in q\_core in the initial loading process.

### get\_qc\_val()

This function returns the associated static function from ‘qc’ object for the given key. ‘qc’ was prepared by loading the ‘libq\_core.so’

### q\_add()

This function plays an important role in dynamic compilation. This is the API exposed by q\_core to register the external functions which are not part of “libq\_core.so”. If the function is already registered with q\_core or for statically defined functions (present in ‘qc’), q\_add() never should get called.

Whenever a new function (e.g vvadd\_I4\_I4\_I4) has to register with q\_core, q\_add() should be provided with below arguments

**doth** - header file contents

**dotc** - c file contents (this should have only one exposable function)

**fn\_name** - exposable function name

To work q\_add() with expanders of the operator, q\_add is supported with the specializer output. In general, specializer of the operator returns

**subs** - substitutions that are required to put in template file

**tmpl** - template file name, this template file contains header and C file content in specific format

In this case, q\_add() uses the ‘gen\_code’ utility to get **doth** & **dotc** contents.

**Note:** ‘gen\_code’ utility expects the full path of template file or else if just file name is provided then file with this name should be present in Q\_TMPL\_DIR

With the current design, specializer returns just a template file name. In this case, it is the responsibility of the operator Makefile to copy template file under Q\_TMPL\_DIR

Using the **doth** & **dotc** contents, q\_add() compiles it and prepares the h\_path and so\_path.

Here,

h\_path - Q\_INC\_DIR/<fn\_name>.h

so\_path - Q\_LIB\_DIR/lib<fn\_name>.so

At the end, q\_add() ask load\_lib() to load this newly created h\_path & so\_path

### wrap()

This functions helps to measure the performance of the target function, it provides you the execution time of target function to perform the activity.

This function wraps the target function with timing calculation and returns the wrapped function.

The wrapping activity is in picture only in when qconsts.qc\_trace = true

### q\_core metamethod - \_\_index() and \_\_newindex()

**\_\_index()**

This metamethod is called whenever any key is getting accessed from the q\_core table. Here, It checks

* If key is ‘q\_add’ (a reserved keyword in Q) then it returns the q\_add() function
* else if key is present in function\_lookup table ( a internal data-structure), returns respective dynamically registered function
* else if there is any static function present in qc with this key name, returns it
* else returns nil

**\_\_newindex()**

This metamethod is called whenever any assignment is happening in the q\_core table.

This has effects only in debug mode i.e qconsts.debug = true, else you can not add a new key to q\_core table.

It helps us in below case

Consider a case where you want to debug your generator function of a operator, i.e you want to verify the arguments/values which get passed to C function. You want to achieve this by providing a custom lua verification function instead of falling into C world. This metamethod makes it possible, in debug mode just assign your custom lua verification function to the actual value

E.g q\_core[vvadd\_I4\_I4\_I8] = my\_verify\_function

## Library loading process

When q\_core is loaded first time, it performs below steps

* ffi.cdef the q\_core header file i.e “q\_core.h” from Q\_INC\_DIR
* ffi.load the q\_core “.so” file i.e “libq\_core.so” from Q\_LIB\_DIR
* loads all “.so” from Q\_LIB\_DIR and respective “.h” files from Q\_INC\_DIR except q\_core using add\_libs()

## Support for dynamic compilation

q\_core supports the dynamic compilation with the help function registration process through q\_add() API from q\_core.

As you go on using the operators, respective C functions get registered with q\_core on the first use. So registration is the one time activity. Next time onwards just use the registered function using q\_core.

In this way, DC helps to keep the bare minimal Q traces (library) on the machine.

It works as below

* When you build Q, it creates ‘libq\_core.so’ with minimum required functions/symbols in it.
* For the first time, when you access any Q function, lets say Q.vvadd(vec1, vec2), operator expander registers the respective C function (e.g vvadd\_I4\_I4\_I8) with q\_core with the help of q\_add() API
* Here, expander provides the appropriate arguments to q\_add() as explained in the q\_add() function details
* Once the C function get registered, now onwards call to this function will be served from internal data-structures of the q\_core (considering same luajit instance)
* In each function registration process, a “.so” file and a header will get created in Q\_LIB\_DIR and Q\_INC\_DIR respectively with specific naming format.
* In a new luajit instance, when q\_core is loaded first time, all “.so” files and respective header files will get loaded from Q\_LIB\_DIR and Q\_INC\_DIR respectively.
* This way already registered functions get ready for this instance, no separate registration process is required.

To achieve the DC, below changes are need to be done

**operator expander changes:**

* Need to check whether the symbol (or function) is already registered with q\_core, if not add a call to q\_add() with required args

====================================

-- START: Dynamic compilation

if ( not qc[func\_name] ) then

print("Dynamic compilation kicking in... ")

qc.q\_add(subs, tmpl, func\_name)

end

-- STOP : Dynamic compilation

====================================

**operator Makefile changes:**

* Makefile should have a target for dynamic compilation which will perform below task
  + create the specializer files if required (from specializer template)
  + copy static C files to Q\_BUILD\_DIR/src and respective header files to Q\_BUILD\_DIR/include
* It should not generate C code and compile it to create the “.so” file.

**Q/UTILS/build/Makefile changes:**

* For each operator, this Makefile should refer to the respective DC target from operator’s Makefile

## Things to take care

* Never attempt to dynamically compile for a symbol that exists
* Every dynamically compiled c file should have exactly one function that we want exposed
* q\_add is the only reserved word in qcore. Even if you register a function with that name it will call the original q\_add
* For each header file in “$Q\_ROOT/include” dir (except “q\_core.h”), there should be a respective “.so” file in “$Q\_ROOT/lib” dir
* Convention for header file and respective “.so” file should be as below (except “q\_core.h”), for a function ‘fn\_name’

header file - **<fn\_name>.h**

“.so” file - **lib<fn\_name>.so**

* “libq\_core.so” in “$Q\_ROOT/lib” and “q\_core.h” in “$Q\_ROOT/include” should be present after Q build.