Release notes for the Quick C-- compiler
Release 20140124

The Quick C-- Team

January 24, 2014

Contents

1 Introduction

This document contains information you need to use the Quick C-- compiler. It is divided into several sections, which do different things.

- We provide the target-specific information that is required by the C-- specification.
- We explain how to integrate your front end into the Quick C-- driver. (If you prefer just to use Quick C-- to translate C-- files into assembly language, that's done by `qc--.opt -S` as explained on the man page.)
- Quick C-- is a research prototype; we list some unpleasant ways that it may surprise you.
- We explain how to identify the mysterious error messages that Quick C-- issues when something goes badly wrong, and when and how to send in a bug report.
- We discuss experimental support for multithreading.
- We provide other notes for the latest release of the compiler, as well as old notes for previous releases. The old notes are useful only if you are upgrading from an earlier release. Each release of Quick C-- is numbered with the day of release in YYYYMMDD format.

This document is not complete by itself; you also need to read the man page for `qc--`.

2 Portability (target-specific features)

Since release 20041004, Quick C-- has provided some target-specific information at run time; to get this information try

```
qc-- -e 'Target.explain()'
```

It is also possible to pass a particular back end, e.g.,

```
qc-- -e 'Target.explain(Backend.ppc)'
```

If you want to process this information in a program, you can get it in different form by using Lua 2.5, which is the scripting language of the Quick C-- compiler.

In addition to the information emitted by the compiler, we provide the further information below.

2.1 Intel x86

As of release 20050331, Quick C-- supports only one back end, for Intel x86 running Linux. (Other back ends are in the late experimental stages.)

The x86 back end has these known bugs:

- The `%NaN` operator may produce a quiet NaN even when signalling NaN is called for.

The x86 back end has the following properties:

Supported data types: Quick C-- supports these types:

- Type `bits32` is supported for register variables and for all computation.
- Types `bits16` and `bits8` are supported for memory references, and they work with operators `%sx`, `%zx`, and `%lobits`. In particular, here's what you can do with a narrow variable or memory location:
  - Assign it a compile-time constant expression
  - Assign it the contents of another variable or memory location of the same width
  - Assign it `%lobits` of an expression of type `bits32`.
- Type `bits64` can be stored in memory and registers, passed as parameters, and returned as results. This includes passing to and from gcc-compiled C functions, e.g., as `long long int`. But operators `%sx`, `%zx`, and `%lobits` are not supported for 64-bit values, and actual operations on 64-bit values have to be done by calling out to a C routine.
- Larger bit vectors whose widths are a multiple of 8 can be supported for memory references, variables, and parameters, but the only computation available is assignment, which translates to block copy.
- The type `bits2` is supported for a reference to the floating-point rounding mode.
- Variables of type `bits1` are supported for addition and subtraction with carry and borrow operations. They can also be sign- or zero-extended out to 32 bits, and similarly, one can use `%lobits1` to extract a bit to feed into carry, if desired.

Target metrics: The native word and pointer types are `bits32`. The addressing unit is the 8-bit byte, and the byte order is little-endian:

```
target byteorder little
       memsize 8
       wordsize 32 pointersize 32;
```

All but byte order can be defaulted.

Primitive operators: Release 20050331 supports all the primitives in the C-- specification. For details, run `qc-- -e 'Target.explain()'`.

Integer primitives are supported at 32 bits. Floating-point primitives are nominally supported at 32, 64, and 80 bits, but in fact all floating-point operations are widened to 80 bits, which is the natural width of the x86 floating-point unit. Results from "narrow floating-point arithmetic" therefore may be other than expected.

None of the primitives are implemented in alternate-return form; therefore, it is not possible to use the hardware to recover from such errors as division by zero.

Back-end capabilities: Integer literals are supported in any width up to 64 bits. Floating-point literals are availabe in types `bits32` and `bits64` only.

There should be no difficulty generating code for expressions involving the supported primitives at width 32, plus sign-extended and zero-extended narrow *values* (e.g., references to memory). Assignments of any number of bytes are also supported.

Foreign interface: Quick C-- supports two foreign calling conventions: `foreign "C"` and `foreign "paranoid C"`. Because the C calling convention does not specify where callee-saves registers are saved, it is necessary to use `foreign "paranoid C"` when calling a C function that might be on the stack when the run-time system is active.

It is relatively easy to add new conventions to Quick C-- at need; if you want a custom calling convention, ask.

Hints in calling conventions: Quick C-- supports these three hints:

`"float"`	A floating-point number
`"address"`	A memory address or pointer
`""`	Any non-float, non-pointer data, including signed and unsigned integers, bit vectors, records passed by value, and so on

The x86 platform has no address registers, so the x86 back end should treat `"address"` the same as `""`.

Hints in register declarations: Quick C-- supports the same three hints in global register declarations.

Names of hardware registers: The only hardware register currently supported (as of 20050331) is a 2-bit register named `"IEEE 754 rounding mode"`, which may be used as follows, for example:

```
bits2 rm = "IEEE 754 rounding mode";
```

3 Integrating your front end into the Quick C-- driver

Quick C-- is scripted using version 2.5 of the programming language Lua (see `lua.org`). Most of the benefits of scripting are realized by the developers of the compiler, not by end users, but there is one advantage for end users: you can easily teach the driver about your source files and front end. To do so, you need to tell the driver several things: how to find your front end, when to use it, and what sort of run-time system to link with your code. You can do all this by creating a small Lua file.

Here is an example from the Tiger front end. We begin by creating a Lua table to hold Tiger-related functions and private data:

⟨*tiger.lua*⟩≡
```
Frontends.Tiger = Frontends.Tiger or { }  -- initialize table
```

For our next step, here is a convenience function that makes it easier to find tiger-related files in `/usr/local/lib/tiger`.

⟨*tiger.lua*⟩+≡
```
Frontends.Tiger.dir = "/usr/local/lib/tiger" --- where the front end lives

function Frontends.Tiger.file(name)
  return Frontends.Tiger.dir .. '/' .. name -- not portable
end
```

The `..` symbol means string concatentation.

Our next step is to make sure that the libraries passed to the linker include the Tiger run-time system and standard library.

⟨*tiger.lua*⟩+≡
```
Ld.libs = Ld.libs .. " " .. Frontends.Tiger.file("runtime.o")
                  .. " " .. Frontends.Tiger.file("stdlib.a")
```

Finally, we need to teach the driver that a file with extension `.tig` means a Tiger file, and that it should be translated into a `.c--` file using the Tiger front end. We do this by defining a function `CMD.compilertab[".tig"]`, which is where the driver will look when compiling a file with extension `".tig"`.

⟨*tiger.lua*⟩+≡
```
function CMD.compilertab[".tig"](file)
  local out     = CMD.outfilename(file, ".c--")
  local tigerc  = Frontends.Tiger.file("tigerc")
  local options = ""
  if Frontends.Tiger.unwind then options = "-unwind" end
  CMD.exec(tigerc .. " " .. options .. " " .. file .. " > " .. out)
  return out
end
```

Function `CMD.exec` executes a string as a shell command, where the call to `CMD.outfilename` gives the local variable `out` a file name with extension `.c--`. *Returning* `out` is key, because it is by returning the `.c--` file that we tell the driver to continue translating: it will translate the C-- code into assembly language, then object code.

The `options` variable helps deal with options to be passed to the Tiger compiler. Here, if variable `Frontends.Tiger.unwind` is set, we pass `-unwind`, which tells `tigerc` to compile exceptions using stack unwinding. We can then say on the command line something like

```
qc--.opt tiger.lua Frontends.Tiger.unwind=1 ...
```

Finally, so that we can not only compile Tiger code but also interpret it or prettyprint it, we assign the same function to two more table entries.

⟨*tiger.lua*⟩+≡

```
CMD.interptab[".tig"] = CMD.compilertab[".tig"]
CMD.prettytab[".tig"] = CMD.compilertab[".tig"]
```

4 Unpleasant surprises

Quick C-- is not yet as complete as we would like. Many of the limitations of the code generator are mentioned in Section 2 (Portability), but we highlight some here.

4.1 Floating-point computation

- For the semantics of floating-point literals, we trust the Objective Caml compiler implicitly. This means we don't know exactly what guarantees are provided. If you must get exactly the floating-point literal you want, it is best to emit a hexadecimal literal that produces the right bit-pattern.
- The only rounding modes that are supported are the hardware rounding modes of the target machine. These need to be declared as a special global register variable, e.g.,

  ```
  bits2 System.rounding_mode "IEEE 754 rounding mode";
  ```

  The name "`System.rounding_mode`" is not special; it's the string literal that makes this name refer to the hardware.
- You can't usefully assign to the rounding modes or read them—all you can usefully do is use them in floating-point operations that require rounding modes. (This one might not be too hard to fix.)

4.2 Widths of computation

- The back ends support computation at widths that are at most the natural width of the target machine.

4.3 Poor performance

The compiler generates very bad code. The implementation of the `switch` statement is especially naïve. Code can be improved somewhat by using our peephole optimizer. You can try it out by naming the script `peephole.lua` on the command line, e.g.,

```
qc--.opt peephole.lua -globals hello.c--
```

On October 4, 2004, João Dias fixed the last known bug in the peephole optimizer—but it is still not turned on by default.

5 Mysterious errors messages, bugs, and bug reports

Any compile-time error message that begins with the phrase "`This can't happen`" indicates an internal error in the Quick C-- compiler. In case of such a message, please post an issue to the public Github repository `nrnrnr/qc--`.

Any compile-time error message that begins with the phrase "`Not implemented in qc--`" indicates a feature of C-- that the Quick C-- compiler does not yet support. If you want such a feature supported, please post an issue.

A link-time error that looks something like

```
undefined reference to `Cmm.global_area'
```

means that you have not used the `-globals` flag on any of your compilations. If you get a link-time error which indicates that `Cmm.global_area` is multiply defined, it that you have used the `-globals` flag on more than one compilation.

A link-time error that looks something like

```
undefined reference to `Cmm.globalsig.RMRFFdVEETFHHBeVALWMMEBYUE'
```

means either that you have not used the `-globals` flag on any of your compilations (in which case you should also get the error above), or else you have different global-variable declarations in different compilation units.

6 Experimental features

Thread support is not yet working, but it should be in place by the end of June.

6.1 Thread support

The front-end runtime can create a C-- thread using `Cmm_CreateStack`, then get C-- to transfer control to it using `cut to`.

This interface will certainly change once we have support for a stack-limit check.

`Cmm_Cont *Cmm_CreateStack(Cmm_CodePtr f, Cmm_DataPtr x, void *stack, unsigned n)` returns a C-- continuation that, when `cut to`, will execute the C-- call `f(x)` on the stack `stack`.

- The parameter `f` must be the address of a C-- procedure that takes exactly one argument, which is a value of the native pointer type. To pass any other `f` to `Cmm_CreateStack` is an *unchecked* run-time error.

  It is a checked *run-time error for the procedure addressed by* `f` *to return*— this procedure should instead finish execution with a `cut to`.

- When queried using the C-- run-time interface, a continuation returned by `Cmm_CreateStack` looks like a stack with one activation. That activation makes the two parameters `f` and `x` visible through `Cmm_FindLocalVar`; these parameters can be changed using the run-time interface (for example, if a garbage collection intervenes between the time the continuation is created and it is used).

- When a continuation returned by `Cmm_CreateStack` is `cut to`, it is as if the stack makes a tail call `jump f(x)`. In particular, the activation of `f` now appears as the oldest activation on the stack. As noted, it is a checked run-time error for this activation to return.

- The parameter `stack` is the address of the stack, which is `n` bytes in size. After calling `Cmm_CreateStack`, the stack belongs to C--, so it is an *unchecked* run-time error for the front end to read or write any part of this stack except through `stackdata` areas in active procedures (or through pointers provided by the C-- run-time interface).

  Although this experimental thread support does no checking for stack overflow, we nevertheless need the size of the stack as well as its address, because it is a machine-dependent property whether the oldest activation goes at the high end or the low end of the stack.

To implement threads, a front end will typically allocate a large thread-control block to represent a thread, and the C-- stack will be just a part of this block. The rest of the block may contain a C-- continuation for the thread, thread-local data, the priority of the thread, links to other threads, and so on. All of this information is outside the purview of C--, however.

7 Notes for release 20140124

This release contains a number of small improvements (plus a new register allocator) largely made by John Dias in 2006 and 2007. The project was tabled shortly after these improvements were made. The 2014 release is a result of getting the code off of CVS and onto Github. Some tests pass (but all the tiger tests fail, so probably the release is not configured correctly).

Here are some other helpful notes carried forward from previous releases:

- Don't overlook the `-globals` option (described in the man page), which is used to guarantee consistency of C-- global-variable declarations across separately compiled units.
- Quick C-- has a very simple peephole optimizer. It can reduce the size of a program by 30% or so, but it does not work on all programs. You can try it out by running

```
qc--.opt Backend.x86.improve=Optimize.improve ...
```

- Quick C-- has a graph-coloring register allocator, but because it is slow, it is turned off by default. To turn it on, run

```
qc--.opt Backend.x86.ralloc=Ralloc.color ...
```

- There are a few inconsistencies between the run-time interface in the specification and the one implemented in the Quick C-- interpreter.
- Normally you want to run `qc--.opt`, but if Objective Caml cannot build a native-code binary on your system, you can use `qc--` instead.

8 Old notes for older releases

These notes are in reverse chronological order. You need to read them only if you are upgrading from an older release and want to know what has changed.

8.1 Notes for release 20050331

Release 20050331 incorporates a major change in Quick C--'s representation of a control-flow graph. We hope that the new representation, unlike its five predecessors, will not be a bug factory. With the new release, the automated facility for isolating optimization bugs is not yet available. This facility will be restored in a future release.

8.2 Notes for release 20040522

We've added some support for floating-point literals.

8.3 Notes for release 20040514

We now have a very simple peephole optimizer. It can reduce the size of a program by 30% or so. You can try it out by running

```
qc--.opt Backend.x86.improve=Optimize.improve ...
```

If you find a program that breaks the optimizer, please send it to us.

8.4 Notes for release 20040504

Here's what you can do with a narrow variable or memory location:

- Assign it a compile-time constant expression
- Assign it the contents of another variable or memory location of the same width
- Assign it `%lobits` of an expression whose width is the natural width of the target machine

8.5 Notes for release 20040501

We've added limited support for 64-bit variables on 32-bit platforms. You can perform the following operations on a 64-bit C-- variable:

- Assign it from a 64-bit literal
- Load it from memory
- Store it to memory
- Pass it to a C function
- Receive it from a C function

For the moment, actual operations on 64-bit values have to be done by calling out to a C compiler. Here, for example, is a C-- procedure that tests 64-bit add:

```
target byteorder little;

import ladd, printf;
export main;

foreign "C" main(bits32 argc, "address" bits32 argv) {
  bits64 n, m;
  n = 0x80000000::bits64;
  m = 0x80000000::bits64;
  bits64 sum;
  sum = foreign "C" ladd(n, m);
  foreign "C" printf(format, n, m, sum);
  foreign "C" return(0);
}

section "data" {
  format: bits8[] "sum of %016llx\nand    %016llx\nis     %016llx\n\0";
}
```

Here the imported function `ladd` is defined in C as follows:

```
long long ladd(long long x, long long y) {
  return x + y;
}
```

The 64-bit variables are not actually stored in 32-bit register pairs, but that optimization will probably come in a future release.

8.6 Notes for release 20040427

Quick C-- has a new front end. Error messages should be more informative, and the front end now detects some bugs that were undetected by the old front end.

In case of unforeseen difficulty, it is possible to use the old front end by

```
qc-- -e 'Compile.Old.use()' ...
```

but this option is deprecated and will disappear in a future release. Any problems with the new front end should be reported to `bugs@cminusminus.org`.

8.7 Notes for release 20040405

- Unicode string literals are not implemented
- Integer widening has been added, but not enabled. In order to use widening, you must swap the order of the "widen" and "placevars" phases. Look for the following lines in "luacompile.nw":

```
      Opt.verb(backend.placevars)  or Stages.null
    , Opt.verb(backend.widen)      or Stages.null
```

  The backend "x86w" shows how to construct a backend that does integer widen properly for x86.

8.8 Notes for release 20031021

- The compiler now generates runtime data. The native runtime system now lives in the runtime directory.
- The interpreter interface has been changed slightly to be more consistent with the native runtime interface. However, there are still a few differences between the two interfaces. Interpreter functions take pointers to activation structures where the native runtime takes pointers to activation structures.
- The default register allocator has been changed to DLS. If you need the graph coloring allocator, you can change it on the command line, e.g.,

```
qc-- backend=Backend.x86 backend.ralloc=color ...
```

8.9 Notes for more recent releases

- If two different C-- units have different global-variable declarations, the error manifests as a failure to link.

8.10 Notes for release 20030711

This is the first major release that includes a native-code back end. The release is believed consistent with the pre-2.0 manual (CVS revision 1.75) with exceptions noted below.

- The front end does not implement the `switch` statement.
- The `import` statement requires a type, although this requirement is not documented in the manual.
- The native x86 back end successfully compiles the test suite for the `lcc` C compiler. The back end has significant limitations:
  - Integer variables should be 8, 16, or 32 bits. Integer and logical operations should be on 32-bit values only. The back end supports the operations you would find in a C compiler, but we can try to add others on request.
  - Floating-point operations may be at 32 or 64 bits. 80-bit floating-point is possible but not tested. The back end supports the operations you would find in a C compiler, but we can try to add others on request.
  - Mixed integer and floating-point operations have not been tested thoroughly.
  - The back end does not support multiprecision arithmetic or overflow detection.
- The release also includes back ends for Alpha and MIPS R3000, but these are incomplete and have been used only to test implementations of calling conventions.
- There is as yet no run-time system to go with the native-code compiler. If you need a run-time system, you must use the interpreter.
- Code quality is poor; as a matter of policy, we are postponing work on optimization in order to bring you a run-time system.
- The default register allocator is the `dls` register allocator. This allocator is dramatically faster than our implementation of the graph-coloring register allocator (`color`). If you want, you may invoke the `color` allocator on the command line, as in the following example:

```
qc-- Backend.x86.ralloc=Ralloc.color hello.c--
```