QuickJS Javascript Engine
Table of Contents

1 Introduction ................................................................. 1
   1.1 Main Features .......................................................... 1

2 Usage ......................................................................... 2
   2.1 Installation .............................................................. 2
   2.2 Quick start .............................................................. 2
   2.3 Command line options .................................................. 2
      2.3.1 qjs interpreter ..................................................... 2
      2.3.2 qjsc compiler ...................................................... 3
   2.4 qjscalc application ................................................... 3
   2.5 Built-in tests ........................................................... 3
   2.6 Test262 (ECMAScript Test Suite) ...................................... 4

3 Specifications ............................................................ 5
   3.1 Language support ...................................................... 5
      3.1.1 ES2023 support ..................................................... 5
      3.1.2 ECMA402 ............................................................. 5
      3.1.3 Extensions .......................................................... 5
      3.1.4 Mathematical extensions ........................................ 5
   3.2 Modules ................................................................. 5
   3.3 Standard library ....................................................... 5
      3.3.1 Global objects ..................................................... 6
      3.3.2 std module .......................................................... 6
      3.3.3 os module ........................................................... 9
   3.4 QuickJS C API .......................................................... 12
      3.4.1 Runtime and contexts ............................................ 13
      3.4.2 JSValue ............................................................. 13
      3.4.3 C functions ........................................................ 13
      3.4.4 Exceptions .......................................................... 13
      3.4.5 Script evaluation ................................................ 13
      3.4.6 JS Classes .......................................................... 13
      3.4.7 C Modules .......................................................... 14
      3.4.8 Memory handling ................................................ 14
      3.4.9 Execution timeout and interrupts ............................. 14

4 Internals ................................................................. 15
   4.1 Bytecode ............................................................... 15
   4.2 Executable generation ................................................. 15
      4.2.1 qjsc compiler ..................................................... 15
      4.2.2 Binary JSON ........................................................ 15
   4.3 Runtime ................................................................. 15
      4.3.1 Strings ............................................................. 15
      4.3.2 Objects ............................................................. 15
      4.3.3 Atoms ............................................................... 16
      4.3.4 Numbers ............................................................ 16
      4.3.5 Garbage collection .............................................. 16
      4.3.6 JSValue ............................................................ 16
4.3.7 Function call
4.4 RegExp
4.5 Unicode
4.6 BigInt, BigFloat, BigDecimal
5 License
1 Introduction

QuickJS is a small and embeddable Javascript engine. It supports most of the ES2023 specification\(^1\) including modules, asynchronous generators, proxies and BigInt.

It supports mathematical extensions such as big decimal float float numbers (BigDecimal), big binary floating point numbers (BigFloat), and operator overloading.

1.1 Main Features

- Small and easily embeddable: just a few C files, no external dependency, 210 KiB of x86 code for a simple “hello world” program.
- Fast interpreter with very low startup time: runs the 77000 tests of the ECMAScript Test Suite\(^2\) in less than 2 minutes on a single core of a desktop PC. The complete life cycle of a runtime instance completes in less than 300 microseconds.
- Almost complete ES2023 support including modules, asynchronous generators and full Annex B support (legacy web compatibility). Some features from the upcoming ES2024 specification\(^3\) are also supported.
- Passes nearly 100% of the ECMAScript Test Suite tests when selecting the ES2023 features.
- Compile Javascript sources to executables with no external dependency.
- Garbage collection using reference counting (to reduce memory usage and have deterministic behavior) with cycle removal.
- Mathematical extensions: BigDecimal, BigFloat, operator overloading, bigint mode, math mode.
- Command line interpreter with contextual colorization and completion implemented in Javascript.
- Small built-in standard library with C library wrappers.

\(^1\) https://tc39.es/ecma262/2023
\(^2\) https://github.com/tc39/test262
\(^3\) https://tc39.es/ecma262/
2 Usage

2.1 Installation

A Makefile is provided to compile the engine on Linux or MacOS/X. A preliminary Windows support is available thru cross compilation on a Linux host with the MingGW tools.

   Edit the top of the Makefile if you wish to select specific options then run make.

   You can type make install as root if you wish to install the binaries and support files to /usr/local (this is not necessary to use QuickJS).

   Note: On some OSes atomic operations are not available or need a specific library. If you get related errors, you should either add -latomics in the Makefile LIBS variable or disable CONFIG_ATOMICS in quickjs.c.

2.2 Quick start

   qjs is the command line interpreter (Read-Eval-Print Loop). You can pass Javascript files and/or expressions as arguments to execute them:

     ./qjs examples/hello.js

   qjsc is the command line compiler:

     ./qjsc -o hello examples/hello.js
     ./hello

     generates a hello executable with no external dependency.

2.3 Command line options

2.3.1 qjs interpreter

   usage: qjs [options] [file [args]]

   Options are:

   -h
   --help List options.

   -e EXPR
   --eval EXPR

   Evaluate EXPR.

   -i
   --interactive

   Go to interactive mode (it is not the default when files are provided on the command line).

   -m
   --module Load as ES6 module (default=autodetect). A module is autodetected if the filename extension is .mjs or if the first keyword of the source is import.

   --script Load as ES6 script (default=autodetect).

   --bignum Enable the bignum extensions: BigDecimal object, BigFloat object and the "use math" directive.

   -I file
--include file
  Include an additional file.

  Advanced options are:
  --std    Make the std and os modules available to the loaded script even if it is not a module.
  -d
  --dump   Dump the memory usage stats.
  -q
  --quit   just instantiate the interpreter and quit.

2.3.2 qjsc compiler
usage: qjsc [options] [files]
Options are:
  -c        Only output bytecode in a C file. The default is to output an executable file.
  -e        Output main() and bytecode in a C file. The default is to output an executable file.
  -o output Set the output filename (default = out.c or a.out).
  -N cname  Set the C name of the generated data.
  -m        Compile as Javascript module (default=autodetect).
  -D module_name
            Compile a dynamically loaded module and its dependencies. This option is needed when your code uses the import keyword or the os.Worker constructor because the compiler cannot statically find the name of the dynamically loaded modules.
  -M module_name[,cname]
            Add initialization code for an external C module. See the c_module example.
  -x        Byte swapped output (only used for cross compilation).
  -flto     Use link time optimization. The compilation is slower but the executable is smaller and faster. This option is automatically set when the -fno-x options are used.
  -fno-[eval|string-normalize|regexp|json|proxy|map|typedarray|promise|bigint]
            Disable selected language features to produce a smaller executable file.
  -fbignum  Enable the bignum extensions: BigDecimal object, BigFloat object and the "use math" directive.

2.4 qjscalc application
The qjscalc application is a superset of the qjs command line interpreter implementing a Javascript calculator with arbitrarily large integer and floating point numbers, fractions, complex numbers, polynomials and matrices. The source code is in qjscalc.js. More documentation and a web version are available at http://numcalc.com.

2.5 Built-in tests
Run make test to run the few built-in tests included in the QuickJS archive.
2.6 Test262 (ECMAScript Test Suite)

A test262 runner is included in the QuickJS archive. The test262 tests can be installed in the QuickJS source directory with:

```
  git clone https://github.com/tc39/test262.git test262
  cd test262
  patch -p1 < ../tests/test262.patch
  cd ..
```

The patch adds the implementation specific harness functions and optimizes the inefficient RegExp character classes and Unicode property escapes tests (the tests themselves are not modified, only a slow string initialization function is optimized).

The tests can be run with

```
  make test2
```

The configuration files test262.conf (resp. test262o.conf for the old ES5.1 tests\(^1\)) contain the options to run the various tests. Tests can be excluded based on features or filename.

The file test262_errors.txt contains the current list of errors. The runner displays a message when a new error appears or when an existing error is corrected or modified. Use the -u option to update the current list of errors (or make test2-update).

The file test262_report.txt contains the logs of all the tests. It is useful to have a clearer analysis of a particular error. In case of crash, the last line corresponds to the failing test.

Use the syntax ./run-test262 -c test262.conf -f filename.js to run a single test. Use the syntax ./run-test262 -c test262.conf N to start testing at test number N.

For more information, run ./run-test262 to see the command line options of the test262 runner.

run-test262 accepts the -N option to be invoked from test262-harness\(^2\) thru eshost. Unless you want to compare QuickJS with other engines under the same conditions, we do not recommend to run the tests this way as it is much slower (typically half an hour instead of about 100 seconds).

---

\(^1\) The old ES5.1 tests can be extracted with

```
  git clone --single-branch --branch es5-tests https://github.com/tc39/test262.git test262o
```

\(^2\) https://github.com/bterlson/test262-harness
3 Specifications

3.1 Language support

3.1.1 ES2023 support
The ES2023 specification is almost fully supported including the Annex B (legacy web compatibility) and the Unicode related features.

The following features are not supported yet:
- Tail calls\(^1\)
- WeakRef and FinalizationRegistry objects
- Symbols as WeakMap keys

3.1.2 ECMA402
ECMA402 (Internationalization API) is not supported.

3.1.3 Extensions
- The directive "use strip" indicates that the debug information (including the source code of the functions) should not be retained to save memory. As "use strict", the directive can be global to a script or local to a function.
- The first line of a script beginning with #! is ignored.

3.1.4 Mathematical extensions
The mathematical extensions are fully backward compatible with standard Javascript. See jsbignum.pdf for more information.
- BigDecimal support: arbitrary large floating point numbers in base 10.
- BigFloat support: arbitrary large floating point numbers in base 2.
- Operator overloading.
- The directive "use bigint" enables the bigint mode where integers are BigInt by default.
- The directive "use math" enables the math mode where the division and power operators on integers produce fractions. Floating point literals are BigFloat by default and integers are BigInt by default.

3.2 Modules
ES6 modules are fully supported. The default name resolution is the following:
- Module names with a leading . or .. are relative to the current module path.
- Module names without a leading . or .. are system modules, such as std or os.
- Module names ending with .so are native modules using the QuickJS C API.

3.3 Standard library
The standard library is included by default in the command line interpreter. It contains the two modules std and os and a few global objects.

\(^1\) We believe the current specification of tails calls is too complicated and presents limited practical interests.
3.3.1 Global objects

`scriptArgs`
Provides the command line arguments. The first argument is the script name.

`print(...args)`
Print the arguments separated by spaces and a trailing newline.

`console.log(...args)`
Same as `print()`.

3.3.2 std module
The `std` module provides wrappers to the libc `stdlib.h` and `stdio.h` and a few other utilities.

Available exports:

`exit(n)` Exit the process.

`evalScript(str, options = undefined)`
Evaluate the string `str` as a script (global eval). `options` is an optional object containing the following optional properties:

- `backtrace_barrier` Boolean (default = false). If true, error backtraces do not list the stack frames below the `evalScript`.
- `async` Boolean (default = false). If true, `await` is accepted in the script and a promise is returned.

`loadScript(filename)`
Evaluate the file `filename` as a script (global eval).

`loadFile(filename)`
Load the file `filename` and return it as a string assuming UTF-8 encoding. Return null in case of I/O error.

`open(filename, flags, errorObj = undefined)`
Open a file (wrapper to the libc `fopen()`). Return the FILE object or null in case of I/O error. If `errorObj` is not undefined, set its `errno` property to the error code or to 0 if no error occurred.

`popen(command, flags, errorObj = undefined)`
Open a process by creating a pipe (wrapper to the libc `popen()`). Return the FILE object or null in case of I/O error. If `errorObj` is not undefined, set its `errno` property to the error code or to 0 if no error occurred.

`fdopen(fd, flags, errorObj = undefined)`
Open a file from a file handle (wrapper to the libc `fdopen()`). Return the FILE object or null in case of I/O error. If `errorObj` is not undefined, set its `errno` property to the error code or to 0 if no error occurred.

`tmpfile(errorObj = undefined)`
Open a temporary file. Return the FILE object or null in case of I/O error. If `errorObj` is not undefined, set its `errno` property to the error code or to 0 if no error occurred.

`puts(str)` Equivalent to `std.out.puts(str)`.

`printf(fmt, ...args)`
Equivalent to `std.out.printf(fmt, ...args)`.
sprintf(fmt, ...args)
   Equivalent to the libc sprintf().

in
out
err           Wrappers to the libc file stdin, stdout, stderr.
SEEK_SET
SEEK_CUR
SEEK_END     Constants for seek().

Error
   Enumeration object containing the integer value of common errors (additional error
codes may be defined):
   EINVAL
   EIO
   EACCES
   EEXIST
   ENOSPC
   ENOSYS
   EBUSY
   ENOENT
   EPERM
   EPIPE

strftime(erro)
   Return a string that describes the error errno.

gc()          Manually invoke the cycle removal algorithm. The cycle removal algorithm is auto-
              matically started when needed, so this function is useful in case of specific memory
              constraints or for testing.

getenv(name)  Return the value of the environment variable name or undefined if it is not defined.

setenv(name, value)
   Set the value of the environment variable name to the string value.

unsetenv(name)
   Delete the environment variable name.

getenviren()  Return an object containing the environment variables as key-value pairs.

urlGet(url, options = undefined)
   Download url using the curl command line utility. options is an optional object
   containing the following optional properties:

   binary       Boolean (default = false). If true, the response is an ArrayBuffer instead
                 of a string. When a string is returned, the data is assumed to be UTF-8
                 encoded.
full

Boolean (default = false). If true, return the an object contains the properties `response` (response content), `responseHeaders` (headers separated by CRLF), `status` (status code). `response` is `null` is case of protocol or network error. If `full` is false, only the response is returned if the status is between 200 and 299. Otherwise `null` is returned.

`parseExtJSON(str)`

Parse `str` using a superset of `JSON.parse`. The following extensions are accepted:
- Single line and multiline comments
- unquoted properties (ASCII-only Javascript identifiers)
- trailing comma in array and object definitions
- single quoted strings
- `\f` and `\v` are accepted as space characters
- leading plus in numbers
- octal (`0o` prefix) and hexadecimal (`0x` prefix) numbers

FILE prototype:

`close()` Close the file. Return 0 if OK or `-errno` in case of I/O error.

`puts(str)` Outputs the string with the UTF-8 encoding.

`printf(fmt, ...args)` Formatted printf.

The same formats as the standard C library `printf` are supported. Integer format types (e.g. `%d`) truncate the Numbers or BigInts to 32 bits. Use the `l` modifier (e.g. `%ld`) to truncate to 64 bits.

`flush()` Flush the buffered file.

`seek(offset, whence)` Seek to a give file position (whence is `std.SEEK_*`). `offset` can be a number or a bigint. Return 0 if OK or `-errno` in case of I/O error.

`tell()` Return the current file position.

`tello()` Return the current file position as a bigint.

`eof()` Return true if end of file.

`fileno()` Return the associated OS handle.

`error()` Return true if there was an error.

`clearerr()` Clear the error indication.

`read(buffer, position, length)` Read `length` bytes from the file to the ArrayBuffer `buffer` at byte position (wrapper to the libc `fread`).

`write(buffer, position, length)` Write `length` bytes to the file from the ArrayBuffer `buffer` at byte position (wrapper to the libc `fwrite`).

`getline()` Return the next line from the file, assuming UTF-8 encoding, excluding the trailing line feed.
readAsString(max_size = undefined)
Read max_size bytes from the file and return them as a string assuming UTF-8 encoding. If max_size is not present, the file is read up its end.

getByte()
Return the next byte from the file. Return -1 if the end of file is reached.

putByte(c)
Write one byte to the file.

3.3.3 os module
The os module provides Operating System specific functions:
- low level file access
- signals
- timers
- asynchronous I/O
- workers (threads)

The OS functions usually return 0 if OK or an OS specific negative error code.

Available exports:

open(filename, flags, mode = 0o666)
Open a file. Return a handle or < 0 if error.

O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT
O_EXCL
O_TRUNC
POSIX open flags.

O_TEXT (Windows specific). Open the file in text mode. The default is binary mode.

close(fd)
Close the file handle fd.

seek(fd, offset, whence)
Seek in the file. Use std.SEEK_* for whence. offset is either a number or a bigint. If offset is a bigint, a bigint is returned too.

read(fd, buffer, offset, length)
Read length bytes from the file handle fd to the ArrayBuffer buffer at byte position offset. Return the number of read bytes or < 0 if error.

write(fd, buffer, offset, length)
Write length bytes to the file handle fd from the ArrayBuffer buffer at byte position offset. Return the number of written bytes or < 0 if error.

isatty(fd)
Return true is fd is a TTY (terminal) handle.

ttyGetWinSize(fd)
Return the TTY size as [width, height] or null if not available.
**ttySetRaw(fd)**
Set the TTY in raw mode.

**remove(filename)**
Remove a file. Return 0 if OK or -errno.

**rename(oldname, newname)**
Rename a file. Return 0 if OK or -errno.

**realpath(path)**
Return [str, err] where str is the canonicalized absolute pathname of path and err the error code.

**getcwd()**
Return [str, err] where str is the current working directory and err the error code.

**chdir(path)**
Change the current directory. Return 0 if OK or -errno.

**mkdir(path, mode = 0o777)**
Create a directory at path. Return 0 if OK or -errno.

**stat(path)**
**lstat(path)**
Return [obj, err] where obj is an object containing the file status of path. err is the error code. The following fields are defined in obj: dev, ino, mode, nlink, uid, gid, rdev, size, blocks, atime, mtime, ctime. The times are specified in milliseconds since 1970. lstat() is the same as stat() excepts that it returns information about the link itself.

**S_IFMT**
**S_IFIFO**
**S_IFCHR**
**S_IFDIR**
**S_IFBLK**
**S_IFREG**
**S_IFSOCK**
**S_IFLNK**
**S_ISGID**
**S_ISUID**
Constants to interpret the mode property returned by stat(). They have the same value as in the C system header sys/stat.h.

**utimes(path, atime, mtime)**
Change the access and modification times of the file path. The times are specified in milliseconds since 1970. Return 0 if OK or -errno.

** symlink(target, linkpath)**
Create a link at linkpath containing the string target. Return 0 if OK or -errno.

**readlink(path)**
Return [str, err] where str is the link target and err the error code.

**readdir(path)**
Return [array, err] where array is an array of strings containing the filenames of the directory path. err is the error code.
setReadHandler(fd, func)
Add a read handler to the file handle fd. func is called each time there is data pending for fd. A single read handler per file handle is supported. Use func = null to remove the handler.

setWriteHandler(fd, func)
Add a write handler to the file handle fd. func is called each time data can be written to fd. A single write handler per file handle is supported. Use func = null to remove the handler.

signal(signal, func)
Call the function func when the signal signal happens. Only a single handler per signal number is supported. Use null to set the default handler or undefined to ignore the signal. Signal handlers can only be defined in the main thread.

SIGINT
SIGABRT
SIGFPE
SIGILL
SIGSEGV
SIGTERM POSIX signal numbers.

kill(pid, sig)
Send the signal sig to the process pid.

exec(args[, options])
Execute a process with the arguments args. options is an object containing optional parameters:

block Boolean (default = true). If true, wait until the process is terminated. In this case, exec return the exit code if positive or the negated signal number if the process was interrupted by a signal. If false, do not block and return the process id of the child.

usePath Boolean (default = true). If true, the file is searched in the PATH environment variable.

file String (default = args[0]). Set the file to be executed.

cwd String. If present, set the working directory of the new process.

stdin

stdout

stderr If present, set the handle in the child for stdin, stdout or stderr.

env Object. If present, set the process environment from the object key-value pairs. Otherwise use the same environment as the current process.

uid Integer. If present, the process uid with setuid.

gid Integer. If present, the process gid with setgid.

getpid() Return the current process ID.

waitpid(pid, options)
waitpid Unix system call. Return the array [ret, status]. ret contains -errno in case of error.
WNOHANG  Constant for the options argument of waitpid.

dup(fd)  dup Unix system call.

dup2(oldfd, newfd)  dup2 Unix system call.

pipe()  pipe Unix system call. Return two handles as [read_fd, write_fd] or null in case of error.

sleep(delay_ms)  Sleep during delay_ms milliseconds.

sleepAsync(delay_ms)  Asynchronous sleep during delay_ms milliseconds. Returns a promise. Example:

    await os.sleepAsync(500);

now()  Return a timestamp in milliseconds with more precision than Date.now(). The time origin is unspecified and is normally not impacted by system clock adjustments.

setTimeout(func, delay)  Call the function func after delay ms. Return a handle to the timer.

clearTimeout(handle)  Cancel a timer.

platform  Return a string representing the platform: "linux", "darwin", "win32" or "js".

Worker(module_filename)  Constructor to create a new thread (worker) with an API close to the WebWorkers. module_filename is a string specifying the module filename which is executed in the newly created thread. As for dynamically imported module, it is relative to the current script or module path. Threads normally don’t share any data and communicate between each other with messages. Nested workers are not supported. An example is available in tests/test_worker.js.

The worker class has the following static properties:

parent  In the created worker, Worker.parent represents the parent worker and is used to send or receive messages.

The worker instances have the following properties:

postMessage(msg)  Send a message to the corresponding worker. msg is cloned in the destination worker using an algorithm similar to the HTML structured clone algorithm. SharedArrayBuffer are shared between workers.

Current limitations: Map and Set are not supported yet.

onmessage  Getter and setter. Set a function which is called each time a message is received. The function is called with a single argument. It is an object with a data property containing the received message. The thread is not terminated if there is at least one non null onmessage handler.

3.4 QuickJS C API

The C API was designed to be simple and efficient. The C API is defined in the header quickjs.h.
Chapter 3: Specifications

3.4.1 Runtime and contexts

**JSRuntime** represents a Javascript runtime corresponding to an object heap. Several runtimes can exist at the same time but they cannot exchange objects. Inside a given runtime, no multi-threading is supported.

**JSContext** represents a Javascript context (or Realm). Each JSContext has its own global objects and system objects. There can be several JSContexts per JSRuntime and they can share objects, similar to frames of the same origin sharing Javascript objects in a web browser.

3.4.2 JSValue

JSValue represents a Javascript value which can be a primitive type or an object. Reference counting is used, so it is important to explicitly duplicate (JS_DupValue(), increment the reference count) or free (JS_FreeValue(), decrement the reference count) JSValues.

3.4.3 C functions

C functions can be created with JS_NewCFunction(). JS_SetPropertyFunctionList() is a shortcut to easily add functions, setters and getters properties to a given object.

Unlike other embedded Javascript engines, there is no implicit stack, so C functions get their parameters as normal C parameters. As a general rule, C functions take constant JSValue as parameters (so they don’t need to free them) and return a newly allocated (=live) JSValue.

3.4.4 Exceptions

Exceptions: most C functions can return a Javascript exception. It must be explicitly tested and handled by the C code. The specific JSValue JS_EXCEPTION indicates that an exception occurred. The actual exception object is stored in the JSContext and can be retrieved with JS_GetException().

3.4.5 Script evaluation

Use JS_Eval() to evaluate a script or module source.

If the script or module was compiled to bytecode with qjsc, it can be evaluated by calling js_std_eval_binary(). The advantage is that no compilation is needed so it is faster and smaller because the compiler can be removed from the executable if no eval is required.

Note: the bytecode format is linked to a given QuickJS version. Moreover, no security check is done before its execution. Hence the bytecode should not be loaded from untrusted sources. That’s why there is no option to output the bytecode to a binary file in qjsc.

3.4.6 JS Classes

C opaque data can be attached to a Javascript object. The type of the C opaque data is determined with the class ID (JSClassID) of the object. Hence the first step is to register a new class ID and JS class (JS_NewClassID(), JS_NewClass()). Then you can create objects of this class with JS_NewObjectClass() and get or set the C opaque point with JS_GetOpaque()/JS_SetOpaque().

When defining a new JS class, it is possible to declare a finalizer which is called when the object is destroyed. The finalizer should be used to release C resources. It is invalid to execute JS code from it. A gc_mark method can be provided so that the cycle removal algorithm can find the other objects referenced by this object. Other methods are available to define exotic object behaviors.

The Class ID are globally allocated (i.e. for all runtimes). The JSClass are allocated per JSRuntime. JS_SetClassProto() is used to define a prototype for a given class in a given JSContext. JS_NewObjectClass() sets this prototype in the created object.

Examples are available in quickjs-libc.c.
3.4.7 C Modules
Native ES6 modules are supported and can be dynamically or statically linked. Look at the test_bjson and bjson.so examples. The standard library quickjs-libc.c is also a good example of a native module.

3.4.8 Memory handling
Use JS_SetMemoryLimit() to set a global memory allocation limit to a given JSRuntime.

Custom memory allocation functions can be provided with JS_NewRuntime2().

The maximum system stack size can be set with JS_SetMaxStackSize().

3.4.9 Execution timeout and interrupts
Use JS_SetInterruptHandler() to set a callback which is regularly called by the engine when it is executing code. This callback can be used to implement an execution timeout.

It is used by the command line interpreter to implement a Ctrl-C handler.
4 Internals

4.1 Bytecode
The compiler generates bytecode directly with no intermediate representation such as a parse tree, hence it is very fast. Several optimizations passes are done over the generated bytecode.

A stack-based bytecode was chosen because it is simple and generates compact code.

For each function, the maximum stack size is computed at compile time so that no runtime stack overflow tests are needed.

A separate compressed line number table is maintained for the debug information.

Access to closure variables is optimized and is almost as fast as local variables.

Direct eval in strict mode is optimized.

4.2 Executable generation

4.2.1 qjsc compiler
The qjsc compiler generates C sources from Javascript files. By default the C sources are compiled with the system compiler (gcc or clang).

The generated C source contains the bytecode of the compiled functions or modules. If a full complete executable is needed, it also contains a main() function with the necessary C code to initialize the Javascript engine and to load and execute the compiled functions and modules.

Javascript code can be mixed with C modules.

In order to have smaller executables, specific Javascript features can be disabled, in particular eval or the regular expressions. The code removal relies on the Link Time Optimization of the system compiler.

4.2.2 Binary JSON
qjsc works by compiling scripts or modules and then serializing them to a binary format. A subset of this format (without functions or modules) can be used as binary JSON. The example test_bjson.js shows how to use it.

Warning: the binary JSON format may change without notice, so it should not be used to store persistent data. The test_bjson.js example is only used to test the binary object format functions.

4.3 Runtime

4.3.1 Strings
Strings are stored either as an 8 bit or a 16 bit array of characters. Hence random access to characters is always fast.

The C API provides functions to convert Javascript Strings to C UTF-8 encoded strings. The most common case where the Javascript string contains only ASCII characters involves no copying.

4.3.2 Objects
The object shapes (object prototype, property names and flags) are shared between objects to save memory.

Arrays with no holes (except at the end of the array) are optimized.

TypedArray accesses are optimized.
4.3.3 Atoms
Object property names and some strings are stored as Atoms (unique strings) to save memory and allow fast comparison. Atoms are represented as a 32 bit integer. Half of the atom range is reserved for immediate integer literals from 0 to $2^{31} - 1$.

4.3.4 Numbers
Numbers are represented either as 32-bit signed integers or 64-bit IEEE-754 floating point values. Most operations have fast paths for the 32-bit integer case.

4.3.5 Garbage collection
Reference counting is used to free objects automatically and deterministically. A separate cycle removal pass is done when the allocated memory becomes too large. The cycle removal algorithm only uses the reference counts and the object content, so no explicit garbage collection roots need to be manipulated in the C code.

4.3.6 JSValue
It is a Javascript value which can be a primitive type (such as Number, String, ...) or an Object. NaN boxing is used in the 32-bit version to store 64-bit floating point numbers. The representation is optimized so that 32-bit integers and reference counted values can be efficiently tested.

In 64-bit code, JSValue are 128-bit large and no NaN boxing is used. The rationale is that in 64-bit code memory usage is less critical.

In both cases (32 or 64 bits), JSValue exactly fits two CPU registers, so it can be efficiently returned by C functions.

4.3.7 Function call
The engine is optimized so that function calls are fast. The system stack holds the Javascript parameters and local variables.

4.4 RegExp
A specific regular expression engine was developed. It is both small and efficient and supports all the ES2023 features including the Unicode properties. As the Javascript compiler, it directly generates bytecode without a parse tree.

Backtracking with an explicit stack is used so that there is no recursion on the system stack. Simple quantifiers are specifically optimized to avoid recursions.

The full regexp library weights about 15 KiB (x86 code), excluding the Unicode library.

4.5 Unicode
A specific Unicode library was developed so that there is no dependency on an external large Unicode library such as ICU. All the Unicode tables are compressed while keeping a reasonable access speed.

The library supports case conversion, Unicode normalization, Unicode script queries, Unicode general category queries and all Unicode binary properties.

The full Unicode library weights about 45 KiB (x86 code).
4.6 BigInt, BigFloat, BigDecimal

BigInt, BigFloat and BigDecimal are implemented with the libbf library\textsuperscript{1}. It weights about 90 KiB (x86 code) and provides arbitrary precision IEEE 754 floating point operations and transcendental functions with exact rounding.

\textsuperscript{1} https://bellard.org/libbf
5 License

QuickJS is released under the MIT license.

Unless otherwise specified, the QuickJS sources are copyright Fabrice Bellard and Charlie Gordon.