QuickJS Javascript Engine

Table of Contents

1 Introduction

QuickJS is a small and embeddable Javascript engine. It supports the ES2019 specification including modules, asynchronous generators and proxies.

It optionally supports mathematical extensions such as big integers (BigInt), big floating point numbers (BigFloat) and operator overloading.

1.1 Main Features

- Small and easily embeddable: just a few C files, no external dependency, 180 KiB of x86 code for a simple "hello world" program.
- Fast interpreter with very low startup time: runs the 56000 tests of the ECMAScript Test Suite[1] in about 100 seconds on a single core of a desktop PC. The complete life cycle of a runtime instance completes in less than 300 microseconds.
- Almost complete ES2019 support including modules, asynchronous generators and full Annex B support (legacy web compatibility).
- Passes 100% of the ECMAScript Test Suite tests.
- Can 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: BigInt, 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://github.com/tc39/test262

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).

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.

`qjsbn` and `qjscbn` are the corresponding interpreter and compiler with the mathematical extensions:

```
./qjsbn examples/pi.js 1000
```

displays 1000 digits of PI.

```
./qjsbnc -o pi examples/pi.js
./pi 1000
```

compiles and executes the PI program.

2.3 Command line options

2.3.1 `qjs` interpreter

`usage: qjs [options] [files]`

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 if .mjs file extension).

Advanced options are:

`-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 if `.mjs` extension).

`-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]`
Disable selected language features to produce a smaller executable file.

2.4 `qjscalc` application

The `qjscalc` application is a superset of the `qjsbn` 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.

For reference, the full test262 tests are provided in the archive `qjs-tests-yyyy-mm-dd.tar.xz`. You just need to untar it into the QuickJS source code directory.

Alternatively, the test262 tests can be installed with:

```
git clone https://github.com/tc39/test262.git test262
cd test262
git checkout 94b1e80ab3440413df916cd56d29c5a2fa2ac451
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
```

For more information, run `./run-test262` to see the options of the test262 runner. The configuration files `test262.conf` and `test262bn.conf` contain the options to run the various tests.

3 Specifications

3.1 Language support

3.1.1 ES2019 support

The ES2019 specification[1] is almost fully supported including the Annex B (legacy web compatibility) and the Unicode related features. The following features are not supported yet:

- Realms (althougth the C API supports different runtimes and contexts)
- Tail calls[2]

3.1.2 JSON

The JSON parser is currently more tolerant than the specification.

3.1.3 ECMA402

ECMA402 (Internationalization API) is not supported.

3.1.4 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.

3.1.5 Mathematical extensions

The mathematical extensions are available in the `qjsbn` version and are fully backward compatible with standard Javascript. See `jsbignum.pdf` for more information.

- The `BigInt` (big integers) TC39 proposal is supported.
- `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] https://tc39.github.io/ecma262/

[2] 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)`
: Evaluate the string `str` as a script (global eval).

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

`Error(errno)`
: `std.Error` constructor. Error instances contain the field `errno` (error code) and `message` (result of `std.Error.strerror(errno)`).

  The constructor contains the following fields:

  `EINVAL`

  `EIO`

  `EACCES`

  `EEXIST`

  `ENOSPC`

  `ENOSYS`

  `EBUSY`

  `ENOENT`

  `EPERM`

  `EPIPE` Integer value of common errors (additional error codes may be defined).

  `strerror(errno)`
  : Return a string that describes the error `errno`.

`open(filename, flags)`
: Open a file (wrapper to the libc `fopen()`). Throws `std.Error` in case of I/O error.

`tmpfile()`
: Open a temporary file. Throws `std.Error` in case of I/O error.

`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().

`global` Reference to the global object.

`gc()` Manually invoke the cycle removal algorithm. The cycle removal algorithm is automatically 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.

FILE prototype:

`close()` Close the file.

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

`printf(fmt, ...args)`
Formatted printf, same formats as the libc printf.

`flush()` Flush the buffered file.

`seek(offset, whence)`
Seek to a give file position (whence is `std.SEEK_*`). Throws a `std.Error` in case of I/O error.

`tell()` Return the current file position.

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

`fileno()` Return the associated OS handle.

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

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

`getline()`
Return the next line from the file, assuming UTF-8 encoding, excluding the trailing line feed.

`getByte()`
Return the next byte from the file.

`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

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`.

`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 < 0 if error.

`rename(oldname, newname)`
: Rename a file. Return 0 if OK or < 0 if error.

`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 hander.

`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 hander.

`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.

`SIGINT`

`SIGABRT`

`SIGFPE`

`SIGILL`

`SIGSEGV`

`SIGTERM` POSIX signal numbers.

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

`clearTimer(handle)`
: Cancel a timer.

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

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`.

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, similary 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 explicitely 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`s 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 explicitely tested and handled by the C code. The specific `JSValue JS_EXCEPTION` indicates that an exception occured. 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`, `JS_EvalBinary()` achieves the same result. 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. 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 `js_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 `js_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 developped. It is both small and efficient and supports all the ES2019 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 quantizers are specifically optimized to avoid recursions.

Infinite recursions coming from quantizers with empty terms are avoided.

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

4.5 Unicode

A specific Unicode library was developped so that there is no dependency on an external large Unicode library such as ICU. All the Unicode tables are compressed while keeping a reasonnable 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 and BigFloat

BigInt and BigFloat are implemented with the `libbf` library[1]. It weights about 60 KiB (x86 code) and provides arbitrary precision IEEE 754 floating point operations and transcendental functions with exact rounding.

[1] https://bellard.org/libbf

5 License

QuickJS is released under the MIT license.