Duktape Programmer's Guide

Introduction

Version: 0.10.0 (2014-04-13)

Document scope

This guide provides an introduction to using Duktape in your programs. Once you're familiar with the basics, there is a concise API reference for looking up API details.

This document doesn't cover Duktape internals (see the Duktape repo if you wish to tinker with them).

What is Duktape?

Duktape is an embeddable Ecmascript E5/E5.1 engine with a focus on portability and compact footprint. By integrating Duktape into your C/C++ program you can easily extend its functionality through scripting. You can also build the main control flow of your program in Ecmascript and use fast C code functions to do heavy lifting.

The terms Ecmascript and Javascript are often considered more or less equivalent, although Javascript and its variants are technically just one environment where the Ecmascript language is used. The line between the two is not very clear in practice: even non-browser Ecmascript environments often provide some browser-specific built-ins. Duktape is no exception, and provides the commonly used `print()` and `alert()` built-ins. Even so, we use the term Ecmascript throughout to refer to the language implemented by Duktape.

Conformance

Duktape conforms to the following Ecmascript specifications:

- Edition 5 (E5)
- Edition 5.1 (E5.1) (as HTML)

The upcoming Ecmascript Edition 6 standard is not yet final and Duktape has no support for its features.

Features

Besides standard Ecmascript features, Duktape has the following additional features (some are visible to applications, while others are internal):

- Additional built-ins: `print()` and `alert()` borrowed from browsers, Duktape specific built-ins in the `Duktape` global object
- Extended types: custom "buffer" and "pointer" types, extended string type which supports arbitary binary strings and non-BMP strings (standard Ecmascript only supports 16-bit codepoints)
- Combined reference counting and mark-and-sweep garbage collection, with finalizers and emergency garbage collection (you can also build with just reference counting or mark-and-sweep)
- Coroutine support
- Tail call support
- Built-in regular expression engine with no platform dependencies

- Built-in Unicode support with no platform dependencies
- Built-in number parsing and formatting with no platform dependencies
- Additional custom JSON formats (JSONX and JSONC)
- Very lightweight built-in logging framework available for both C and Ecmascript code

Goals

Compliance. Ecmascript E5/E5.1 and real world compliance. Ecmascript compliance requires regular expression and Unicode support.

Portability. Minimal system dependencies are nice when porting, so Duktape depends on very few system libraries. For example, number formatting and parsing, regular expressions, and Unicode are all implemented internally by Duktape. One of the few dependencies that cannot be fully eliminated is system date/time integration. This is confined to the implementation of the `Date` built-in.

Easy C interface. The interface between Duktape and C programs should be natural and error-tolerant. As a particular issue, string representation should be UTF-8 with automatic NUL terminators to match common C use.

Small footprint. Code and data footprint should be as small as possible, even for small programs. This is more important than performance, as there are already several very fast engines but fewer very compact, portable engines.

Reasonable performance. Small footprint (and portability, to some extent) probably eliminates the possibility of a competitive JIT-based engine, so there is no practical way of competing with very advanced JIT-based engines like SpiderMonkey (and its optimized variants) or Google V8. Performance should still be reasonable for typical embedded programs. Lua is a good benchmark in this respect. (Adding optional, modular support for JITing or perhaps off-line compilation would be nice.)

ASCII string performance. It's important that operations dealing with plain ASCII strings be very fast: ASCII dominates most embedded use. Operations dealing with non-ASCII strings need to perform reasonably but are not critical. This is a necessary trade-off: using C-compatible strings means essentially using UTF-8 string representation which makes string indexing and many other operations slower than with fixed size character representations. It's still important to support common idioms like iterating strings sequentially (in either direction) efficiently.

Document organization

Getting started guides you through downloading, compiling, and integrating Duktape into your program. It also provides concrete examples of how you can integrate scripting capabilities into your program.

Programming model and Stack types discuss core Duktape concepts such as heap, context, value stacks, Duktape API, and Duktape/C functions. Duktape types are discussed in detail.

Duktape specific Ecmascript features are discussed in multiple sections: Type algorithms (for custom types), Duktape built-ins (additional built-ins), Custom behavior (behavior differing from standard), Custom JSON formats, Error objects (properties and traceback support), Function objects (properties), Finalization, and Coroutines.

Compiling describes how to compile Duktape in detail, covering in particular available feature defines. Performance provides a few Duktape-specific tips for improving performance and avoiding performance pitfalls. Portability covers platform and compiler specific issues and other portability issues. Compatibility discussed Duktape's compatibility with Ecmascript dialects, extensions, and frameworks. Limitations summarizes currently known limitations and provides possible workarounds.

Comparison to Lua discusses some differences between Lua and Duktape; it may be useful reading if you're already familiar with Lua.

Getting started

Downloading

Download the source distributable from the Download page.

Command line tool

Unpack the distributable:

```
$ cd /tmp
$ tar xvfJ duktape-<version>.tar.xz
```

Compile the command line tool using the provided Makefile (your system needs to have `readline` and `ncurses` library and headers installed):

```
# NOTE: ensure you have readline and curses library and headers installed
$ cd /tmp/duktape-<version>/
$ make -f Makefile.cmdline
```

NOTE: The command line tool requires `readline` by default. If it's not available or there are other portability issues, try enabling the `-DDUK_CMDLINE_BAREBONES` compile option in the Makefile. The option minimizes any non-portable dependencies.

You can now run Ecmascript code interactively:

```
$ ./duk
((o) Duktape 0.9.0
duk> print('Hello world!')
Hello world!
= undefined
```

You can also run Ecmascript code from a file which is useful for playing with features and algorithms. As an example, create `fib.js`:

```
// fib.js
function fib(n) {
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    return fib(n-1) + fib(n-2);
}

function test() {
    var res = [];
    for (i = 0; i < 20; i++) {
        res.push(fib(i));
    }
    print(res.join(' '));
}

test();
```

Test the script from the command line:

```
$ ./duk fib.js
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
Cleaning up...
```

Integrating Duktape into your program

The command line tool is simply an example of a program which embeds Duktape. Embedding Duktape into your program is very simple: just add `duktape.c` and `duktape.h` to your build, and call the Duktape API from elsewhere in your program.

The distributable contains a very simple example program, `hello.c`, which illustrates this process. Compile the test program e.g. as (see Compiling for compiler option suggestions):

```
$ cd /tmp/duktape-<version>/
$ gcc -std=c99 -o hello -Isrc/ src/duktape.c examples/hello/hello.c -lm
```

The test program creates a Duktape context and uses it to run some Ecmascript code:

```
$ ./hello
Hello world!
2+3=5
```

Because Duktape is an embeddable engine, you don't need to change the basic control flow of your program. The basic approach is:

- Create a Duktape context e.g. in program initialization (or even on-demand when scripting is needed). Usually you would also load your scripts during initialization, though that can also be done on-demand.
- Identify points in your code where you would like to use scripting and insert calls to script functions there.
- To make a script function call, first push call arguments to the Duktape context's value stack using the Duktape API. Then, use another Duktape API call to initiate the actual call.
- Once script execution is finished, control is returned to your program (the API call returns) and a return value is left on the

Duktape context's value stack for C code to consume.

Let's look at a simple example program. The program reads in a line using a C mainloop, calls an Ecmascript helper to transform the line, and prints out the transformed line. The line processing function can take advantage of Ecmascript goodies like regular expressions, and can be easily modified without recompiling the C program.

The script code will be placed in `process.js`. The example line processing function converts a plain text line into HTML, and automatically bolds text between stars:

```
// process.js
function processLine(line) {
    return line.trim()
        .replace(/[<>&"'\u0000-\u001F\u007E-\uFFFF]/g, function(x) {
            // escape HTML characters
            return '&#' + x.charCodeAt(0) + ';'
         })
        .replace(/\*(.*?)\*/g, function(x, m) {
            // automatically bold text between stars
            return '<b>' + m + '</b>';
         });
}
```

The C code, `processlines.c` initializes a Duktape context, compiles the script, then proceeds to process lines from `stdin`, calling `processLine()` for every line:

```c
/* processlines.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "duktape.h"

int main(int argc, const char *argv[]) {
    duk_context *ctx = NULL;
    char line[4096];
    char idx;
    int ch;

    ctx = duk_create_heap_default();
    if (!ctx) { exit(1); }

    duk_eval_file(ctx, "process.js");
    duk_pop(ctx);  /* pop eval result */

    memset(line, 0, sizeof(line));
    idx = 0;
    for (;;) {
        if (idx >= sizeof(line)) { exit(1); }

        ch = fgetc(stdin);
        if (ch == 0x0a) {
            line[idx++] = '\0';

            duk_push_global_object(ctx);
            duk_get_prop_string(ctx, -1 /*index*/, "processLine");
            duk_push_string(ctx, line);
            duk_call(ctx, 1 /*nargs*/);
            printf("%s\n", duk_to_string(ctx, -1));
            duk_pop(ctx);

            idx = 0;
        } else if (ch == EOF) {
            break;
        } else {
            line[idx++] = (char) ch;
        }
    }

    duk_destroy_heap(ctx);

    exit(0);
}
```

Let's look at the Duktape specific parts of the example code line by line. Here we need to gloss over some details for brevity, see Programming model for a detailed discussion:

```
ctx = duk_create_heap_default();
if (!ctx) { exit(1); }
```

First we create a Duktape context. A context allows us to exchange values with Ecmascript code by pushing and popping values to the **value stack**. Most calls in the Duktape API operate with the value stack, pushing, popping, and examining values on the stack.

```
duk_eval_file(ctx, "process.js");
duk_pop(ctx);  /* pop eval result */
```

The first call reads in `process.js` and evaluates its contents. The script registers the `processLine()` function into the Ecmascript global object. The result of the evaluation is pushed on top of the value stack. Here we don't need the evaluation result, so we pop the value off the stack.

```
duk_push_global_object(ctx);
duk_get_prop_string(ctx, -1 /*index*/, "processLine");
```

The first call pushes the Ecmascript global object to the value stack. The second call looks up `processLine` property of the global object. The script in `process.js` has registered a function into the global object with this name. The `-1` argument is an index to the value stack; negative values refer to stack elements starting from the top, so `-1` refers to the topmost element of the stack, the property lookup result.

```
duk_push_string(ctx, line);
```

Pushes the string pointed to by `line` to the value stack. The string length is automatically determined by scanning for a NUL terminator (same as `strlen()`). Duktape makes a copy of the string when it is pushed to the stack, so the `line` buffer can be freely modified when the call returns.

```
duk_call(ctx, 1 /*nargs*/);
```

At this point the value stack contains: the global object, the processLine function, and the `line` string. This line calls an Ecmascript function with the specified number of arguments; here the argument count is 1. The target function is expected to reside just before the argument list on the value stack. After the API call returns, the arguments have been replaced with a single return value, so the stack contains: the global object and the call result.

```
printf("%s\n", duk_to_string(ctx, -1));
duk_pop(ctx);
```

The `duk_to_string()` call requests Duktape to convert the value stack element at index -1 (the topmost value on the stack, which is the processLine function call result here) to a string, returning a `const char *` pointing to the result. This return value is a read-only, NUL terminated UTF-8 value which C code can use directly as long as the value resides on the value stack. Here we just print out the string. After we've printed the string, we pop the value off the value stack.

```
duk_destroy_heap(ctx);
```

Destroy the Duktape context, freeing all resources held by the context. In our example we left the global object on the value stack on purpose. This call will free the value stack and all references on the value stack. No memory leaks will occur even if the value stack is not empty.

Compile like above:

```
$ gcc -std=c99 -o processlines -Isrc/ src/duktape.c processlines.c -lm
```

Test run:

```
$ echo "I like *Sam & Max*." | ./processlines
I like <b>Sam &#38; Max</b>.
```

Calling C code from Ecmascript (Duktape/C bindings)

The integration example illustrated how C code can call into Ecmascript to do things which are easy in Ecmascript but difficult in C.

Ecmascript also often needs to call into C when the situation is reversed. For instance, while scripting is useful for many things, it is not optimal for low level byte or character processing. Being able to call optimized C helpers allows you to write most of your script logic in nice Ecmascript but call into C for the performance critical parts. Another reason for using native functions is to provide access to native libraries.

To implement a native function you write an ordinary C function which conforms to a special calling convention, the Duktape/C binding. Duktape/C functions take a single argument, a Duktape context, and return a single value indicating either error or number of return values. The function accesses call arguments and places return values through the Duktape context's value stack, manipulated with the Duktape API. We'll go deeper into Duktape/C binding and the Duktape API later on. Example:

```
int my_native_func(duk_context *ctx) {
    double arg = duk_require_number(ctx, 0 /*index*/);
    duk_push_number(ctx, arg * arg);
    return 1;
}
```

Let's look at this example line by line:

```
double arg = duk_require_number(ctx, 0 /*index*/);
```

Check that the number at value stack index 0 (bottom of the stack, first argument to function call) is a number; if not, throws an error and never returns. If the value is a number, return it as a `double`.

```
duk_push_number(ctx, arg * arg);
```

Compute square of argument and push it to the value stack.

```
return 1;
```

Return from the function call, indicating that there is a (single) return value on top of the value stack. You could also return `0` to indicate that no return value is given (in which case Duktape defaults to Ecmascript `undefined`). A negative return value which causes an error to be automatically thrown: this is a shorthand for throwing errors conveniently. Note that you don't need to pop any values off the stack, Duktape will do that for you automatically when the function returns. See Programming model for more details.

We'll use a primality test as an example for using native code to speed up Ecmascript algorithms. More specifically, our test program searches for primes under 1000000 which end with the digits '9999'. The Ecmascript version of the program is:

```
// prime.js

// Pure Ecmascript version of low level helper
function primeCheckEcmascript(val, limit) {
    for (var i = 2; i <= limit; i++) {
        if ((val % i) == 0) { return false; }
    }
    return true;
}

// Select available helper at load time
var primeCheckHelper = (this.primeCheckNative || primeCheckEcmascript);

// Check 'val' for primality
function primeCheck(val) {
    if (val == 1 || val == 2) { return true; }
    var limit = Math.ceil(Math.sqrt(val));
    while (limit * limit < val) { limit += 1; }
    return primeCheckHelper(val, limit);
}

// Find primes below one million ending in '9999'.
function primeTest() {
    var res = [];

    print('Have native helper: ' + (primeCheckHelper !== primeCheckEcmascript));
    for (var i = 1; i < 1000000; i++) {
        if (primeCheck(i) && (i % 10000) == 9999) { res.push(i); }
    }
    print(res.join(' '));
}
```

Note that the program uses the native helper if it's available but falls back to an Ecmascript version if it's not. This allows the Ecmascript code to be used in other containing programs. Also, if the prime check program is ported to another platform where the native version does not compile without changes, the program remains functional (though slower) until the helper is ported. In this case the native helper detection happens when the script is loaded. You can also detect it when the code is actually called which is more flexible.

A native helper with functionality equivalent to `primeCheckEcmascript` is quite straightforward to implement. Adding a program main we get `primecheck.c`:

```c
/* primecheck.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "duktape.h"

static int native_prime_check(duk_context *ctx) {
    int val = duk_require_int(ctx, 0);
    int lim = duk_require_int(ctx, 1);
    int i;

    for (i = 2; i <= lim; i++) {
        if (val % i == 0) {
            duk_push_false(ctx);
            return 1;
        }
    }

    duk_push_true(ctx);
    return 1;
}

int main(int argc, const char *argv[]) {
    duk_context *ctx = NULL;

    ctx = duk_create_heap_default();
    if (!ctx) { exit(1); }

    duk_push_global_object(ctx);
    duk_push_c_function(ctx, native_prime_check, 2 /*nargs*/);
    duk_put_prop_string(ctx, -2, "primeCheckNative");

    duk_eval_file(ctx, "prime.js");
    duk_pop(ctx);  /* pop eval result */

    duk_get_prop_string(ctx, -1, "primeTest");
    duk_call(ctx, 0);
    duk_pop(ctx);

    duk_destroy_heap(ctx);

    exit(0);
}
```

The new calls here are, line by line:

```
int val = duk_require_int(ctx, 0);
int lim = duk_require_int(ctx, 1);
```

These two calls check the two argument values given to the native helper. If the values are not of the Ecmascript number type, an error is thrown. If they are numbers, their value is converted to an integer and assigned to the `val` and `lim` locals. The index 0 refers to the first function argument and index 1 to the second.

```
duk_push_false(ctx);
return 1;
```

Pushes an Ecmascript `false` to the value stack. The C return value 1 indicates that the `false` value is returned to the Ecmascript caller.

```
duk_push_global_object(ctx);
duk_push_c_function(ctx, native_prime_check, 2 /*nargs*/);
duk_put_prop_string(ctx, -2, "primeCheckNative");
```

The first call, like before, pushes the Ecmascript global object to the value stack. The second call creates an Ecmascript `Function` object and pushes it to the value stack. The function object is bound to the Duktape/C function `native_prime_check`: when the Ecmascript function created here is called from Ecmascript, the C function gets invoked. The second argument (2) to the call indicates how many arguments the C function gets on the value stack. If the caller gives fewer arguments, the missing arguments are padded with `undefined`; if the caller gives more arguments, the extra arguments are dropped automatically. Finally, the third call registers the function object into the global object with the name `primeCheckNative` and pops the function value off the stack.

```
duk_get_prop_string(ctx, -1, "primeTest");
duk_call(ctx, 0);
duk_pop(ctx);
```

When we come here the value stack already contains the global object at the stack top. The first line looks up the `primeTest` function from the global object (which was defined by the loaded script). The second line calls the `primeTest` function with zero arguments. The third line pops the call result off the stack; we don't need the return value here.

Compile like above:

```
$ gcc -std=c99 -o primecheck -Isrc/ src/duktape.c primecheck.c -lm
```

Test run:

```
$ time ./primecheck
Have native helper: true
49999 59999 79999 139999 179999 199999 239999 289999 329999 379999 389999
409999 419999 529999 599999 619999 659999 679999 769999 799999 839999 989999

real    0m2.985s
user    0m2.976s
sys     0m0.000s
```

Because most execution time is spent in the prime check, the speed-up compared to plain Ecmascript is significant. You can check this by editing `prime.js` and disabling the use of the native helper:

```
// Select available helper at load time
var primeCheckHelper = primeCheckEcmascript;
```

Re-compiling and re-running the test:

```
$ time ./primecheck
Have native helper: false
49999 59999 79999 139999 179999 199999 239999 289999 329999 379999 389999
409999 419999 529999 599999 619999 659999 679999 769999 799999 839999 989999

real    0m23.609s
user    0m23.573s
sys     0m0.000s
```

Programming model

(This section is under work.)

Overview

Programming with Duktape is quite straightforward:

- Add Duktape source (`duktape.c`) and header (`duktape.h`) to your build.
- Create a Duktape **heap** (a garbage collection region) and an initial **context** (essentially a thread handle) in your program.
- Load the necessary Ecmascript script files, and register your Duktape/C functions. Duktape/C functions are C functions you can call from Ecmascript code for better performance, bindings to native libraries, etc.
- Use the Duktape API to call Ecmascript functions whenever appropriate. Duktape API is used to pass values to and from functions. Values are converted between their C representation and the Duktape internal (Ecmascript compatible) representation.
- Duktape API is also used by Duktape/C functions (called from Ecmascript) to access call arguments and to provide return values.

Let's look at all the steps and their related concepts in more detail.

Heap and context

A Duktape **heap** is a single region for garbage collection. A heap is used to allocate storage for strings, Ecmascript objects, and other variable size, garbage collected data. Objects in the heap have an internal heap header which provides the necessary information for reference counting, mark-and-sweep garbage collection, object finalization, etc. Heap objects can reference each other, creating a reachability graph from a garbage collection perspective. For instance, the properties of an Ecmascript object reference both the keys and values of the object's property set. You can have multiple heaps, but objects in different heaps cannot reference each other directly; you need to use serialization to pass values between heaps.

A Duktape **context** is an Ecmascript "thread of execution" which lives in a certain Duktape heap. A context is represented by a `duk_context *` in the Duktape API, and is associated with an internal Duktape coroutine (a form of a co-operative thread). The context handle is given to almost every Duktape API call, and allows the caller to interact with the **value stack** of the Duktape coroutine: values can be inserted and queries, functions can be called, and so on.

Each coroutine has a **call stack** which controls execution, keeping track of function calls, native or Ecmascript, within the Ecmascript engine. Each coroutine also has a **value stack** which stores all the Ecmascript values of the coroutine's active call stack. The value stack always has an active **stack frame** for the most recent function call (when no function calls have been made, the active stack frame is the value stack as is). The Duktape API calls operate almost exclusively in the currently active stack frame. A coroutine also has an internal **catch stack** which is used to track error catching sites established using e.g. `try-catch-finally` blocks. This is not visible to the caller in any way at the moment.

Multiple contexts can share the same Duktape **heap**. In more concrete terms this means that multiple contexts can share the same garbage collection state, and can exchange object references safely. Contexts in different heaps cannot exchange direct object references; all values must be serialized in one way or another.

Almost every API call provided by the Duktape API takes a context pointer as its first argument: no global variables or states are used, and there are no restrictions on running multiple, independent Duktape heaps and contexts at the same time. There are multi-threading restrictions, however: only one native thread can execute any code within a single heap at any time.

To create a Duktape heap and an initial context inside the heap, you can simply use:

```
duk_context *ctx = duk_create_heap_default();
if (!ctx) { exit(1); }
```

If you wish to provide your own memory allocation functions and a fatal error handler function (recommended), use:

```
duk_context *ctx = duk_create_heap(my_alloc,
                                   my_realloc,
                                   my_free,
                                   my_udata,
                                   my_fatal);
if (!ctx) { exit(1); }
```

To create additional contexts inside the same heap:

```
duk_context *new_ctx;

(void) duk_push_thread(ctx);
new_ctx = duk_get_context(ctx, -1 /*index*/);
```

Contexts are automatically garbage collected when they become unreachable. This also means that if your C code holds a `duk_context *`, the corresponding Duktape coroutine MUST be reachable from a garbage collection point of view.

A heap must be destroyed explicitly when the caller is done with it:

```
duk_destroy_heap(ctx);
```

This frees all heap objects allocated, and invalidates any pointers to such objects. In particular, if the calling program holds string pointers to values which resided on the value stack of a context associated with the heap, such pointers are invalidated and must never be dereferenced after the heap destruction call returns.

Call stack and catch stack (of a context)

The call stack of a context is not directly visible to the caller. It keeps track of the chain of function calls, either C or Ecmascript, currently executing in a context. The main purpose of this book-keeping is to facilitate the passing of arguments and results between function callers and callees, and to keep track of how the value stack is divided between function calls. The call stack also allows Duktape to construct a traceback for errors.

Closely related to the call stack, Duktape also maintains a catch stack for keeping track of current error catching sites established using e.g. `try-catch-finally`. The catch stack is even less visible to the caller than the call stack.

Because Duktape supports tail calls, the call stack does not always accurately represent the true call chain: tail calls will be "squashed" together in the call stack.

> **NOTE:** Don't confuse with the C stack.

Value stack (of a context) and value stack index

The value stack of a context is an array of tagged type values related to the current execution state of a coroutine. The tagged types used are: `undefined`, `null`, boolean, number, string, object, buffer, and pointer. For a detailed discussion of the available tagged types, see Types.

The value stack is divided between the currently active function calls (activations) on the coroutine's call stack. At any time, there is an active stack frame which provides an origin for indexing elements on the stack. More concretely, at any time there is a **bottom** which is referred to with the index zero in the Duktape API. There is also a conceptual **top** which identifies the stack element right above the highest currently used element. The following diagram illustrates this:

```
 Value stack
 of 10 entries
 (absolute indices)

.----.
| 15 |
| 14 |
| 13 |
| 12 |      Active stack frame (indices
| 11 |      relative to stack bottom)
| 10 |
|  9 |      .---.
|  8 |      | 5 |   API index 0 is bottom (at value stack index 3).
|  7 |      | 4 |
|  6 |      | 3 |   API index 5 is highest used (at value stack index 8).
|  5 |      | 2 |
|  4 |      | 1 |   Stack top is 6 (relative to stack bottom).
|  3 | <--- | 0 |
|  2 |      `---'
|  1 |
|  0 |
`----'
```

There is no direct way to refer to elements in the internal value stack: Duktape API always deals with the currently active stack frame. Stack frames are shown horizontally throughout the documentation for space reasons. For example, the active stack frame in the figure above would be shown as:

A **value stack index** is a signed integer index used in the Duktape API to refer to elements in currently active stack frame, relative to the current frame bottom.

Non-negative (>= 0) indices refer to stack entries in the current stack frame, relative to the frame bottom:

Negative (< 0) indices refer to stack entries relative to the top:

The special constant `DUK_INVALID_INDEX` is a negative integer which denotes an invalid stack index. It can be returned from API calls and can also be given to API calls to indicate a "no value".

The **value stack top** (or just "top") is the non-negative index of an imaginary element just above the highest used index. For instance, above the highest used index is 5, so the stack top is 6. The top indicates the current stack size, and is also the index of the next element pushed to the stack.

NOTE:

API stack operations are always confined to the current stack frame. There is no way to refer to stack entries below the current frame. This is intentional, as it protects functions in the call stack from affecting each other's values.

NOTE: Don't confuse with the C stack.

Growing the value stack

At any time, the value stack of a context is allocated for a certain maximum number of entries. An attempt to push values beyond the allocated size will cause an error to be thrown, it will **not** cause the value stack to be automatically extended. This simplifies the internal implementation and also improves performance by minimizing reallocations when you know, beforehand, that a certain number of entries will be needed during a function.

When a value stack is created or a Duktape/C function is entered, the value stack is always guaranteed to have space for the call arguments and `DUK_API_ENTRY_STACK` (currently 64) elements. In the typical case this is more than sufficient so that the majority of Duktape/C functions don't need to extend the value stack. Only functions that need more space or perhaps need an input-dependent amount of space need to grow the value stack.

You can extend the stack allocation explicitly with `duk_check_stack()` or (usually more preferably) `duk_require_stack()`. Once successfully extended, you are again guaranteed that the specified number of elements can be pushed to the stack. There is no way to shrink the allocation except by returning from a Duktape/C function.

Consider, for instance, the following function which will uppercase an input ASCII string by pushing uppercased characters one-by-one on the stack and then concatenating the result. This example illustrates how the number of value stack entries required may depend on the input (otherwise this is not a very good approach for uppercasing a string):

```c
/* uppercase.c */
#include <stdio.h>
#include <stdlib.h>
#include "duktape.h"

static int dummy_upper_case(duk_context *ctx) {
    size_t sz;
    const char *val = duk_require_lstring(ctx, 0, &sz);
    size_t i;

    /* We're going to need 'sz' additional entries on the stack. */
    duk_require_stack(ctx, sz);

    for (i = 0; i < sz; i++) {
        char ch = val[i];
        if (ch >= 'a' && ch <= 'z') {
            ch = ch - 'a' + 'A';
        }
        duk_push_lstring(ctx, (const char *) &ch, 1);
    }

    duk_concat(ctx, sz);
    return 1;
}

int main(int argc, char *argv[]) {
    duk_context *ctx;

    if (argc < 2) { exit(1); }

    ctx = duk_create_heap_default();
    if (!ctx) { exit(1); }

    duk_push_c_function(ctx, dummy_upper_case, 1);
    duk_push_string(ctx, argv[1]);
    duk_call(ctx, 1);
    printf("%s -> %s\n", argv[1], duk_to_string(ctx, -1));
    duk_pop(ctx);

    duk_destroy_heap(ctx);
    return 0;
}
```

In addition to user reserved elements, Duktape keeps an automatic internal value stack reserve to ensure all API calls have enough value stack space to work without further allocations. The value stack is also extended and shrunk in somewhat large steps to minimize memory reallocation activity. As a result the internal number of value stack elements available beyond the caller specified

extra varies considerably. The caller does not need to take this into account and should never rely on any additional elements being available.

Ecmascript array index

Ecmascript object and array keys can only be strings. Array indices (e.g. 0, 1, 2) are represented as canonical string representations of the respective numbers. More technically, all canonical string representations of the integers in the range [0, 2**32-1] are valid array indices.

To illustrate the Ecmascript array index handling, consider the following example:

```
var arr = [ 'foo', 'bar', 'quux' ];

print(arr[1]);     // refers to 'bar'
print(arr["1"]);   // refers to 'bar'

print(arr[1.0]);   // refers to 'bar', canonical encoding is "1"
print(arr["1.0"]); // undefined, not an array index
```

Some API calls operating on Ecmascript arrays accept numeric array index arguments. This is really just a short hand for denoting a string conversion of that number. For instance, if the API is given the integer 123, this really refers to the property name "123".

Internally, Duktape tries to avoid converting numeric indices to actual strings whenever possible, so it is preferable to use array index API calls when they are relevant. Similarly, when writing Ecmascript code it is preferable to use numeric rather than string indices, as the same fast path applies for Ecmascript code.

Duktape API

Duktape API is the collection of user callable API calls defined in `duktape.h` and documented in the API reference.

The Duktape API calls are generally error tolerant and will check all arguments for errors (such as `NULL` pointers). However, to minimize footprint, the `ctx` argument is not checked, and the caller MUST NOT call any Duktape API calls with a `NULL` context.

Duktape/C function

A C function with a Duktape/C API signature can be associated with an Ecmascript function object, and gets called when the Ecmascript function object is called. A Duktape/C API function looks as follows:

```
int my_func(duk_context *ctx) {
    duk_push_int(ctx, 123);
    return 1;
}
```

The function gets Ecmascript call argument in the value stack of `ctx`, with `duk_get_top(ctx)` indicating the number of arguments present on the value stack. When creating an Ecmascript function object associated with a Duktape/C function, one can select the desired number of arguments. Extra arguments are dropped and missing arguments are replaced with `undefined`. A function can also be registered as a vararg function (by giving `DUK_VARARGS` as the argument count) in which case call arguments are not modified prior to C function entry.

The function can return one of the following:

- Return value 1 indicates that the value on the stack top is to be interpreted as a return value.
- Return value 0 indicates that there is no explicit return value on the value stack; an `undefined` is returned to caller.

- A negative return value indicates that an error is to be automatically thrown. Error codes named `DUK_RET_xxx` map to specific kinds of errors (do not confuse these with `DUK_ERR_xxx` which are positive values).
- A return value higher than 1 is currently undefined, as Ecmascript doesn't support multiple return values in Edition 5.1. (Values higher than 1 may be taken into to support multiple return values in Ecmascript Edition 6.)

A negative error return value is intended to simplify common error handling, and is an alternative to constructing and throwing an error explicitly with Duktape API calls. No error message can be given; a message is automatically constructed by Duktape. For example:

```
int my_func(duk_context *ctx) {
    if (duk_get_top(ctx) == 0) {
        /* throw TypeError if no arguments given */
        return DUK_RET_TYPE_ERROR;
    }
    /* ... */
}
```

All Duktape/C functions are considered **strict** in the Ecmascript sense. For instance, attempt to delete a non-configurable property using `duk_del_prop()` will cause an error to be thrown. This is the case with a strict Ecmascript function too:

```
function f() {
    'use strict';
    var arr = [1, 2, 3];
    return delete arr.length;  // array 'length' is non-configurable
}

print(f());  // this throws an error because f() is strict
```

As a consequence of Duktape/C function strictness, all Duktape API calls made from inside a Duktape/C call obey Ecmascript strict semantics. However, when API calls are made from outside a Duktape/C function (when the call stack is empty), all API calls obey Ecmascript non-strict semantics, as this is the Ecmascript default.

Also as a consequence of Duktape/C function strictness, the `this` binding given to Duktape/C functions is not coerced as is normal for non-strict Ecmascript functions. An example of how coercion happens in Ecmascript code:

```
function strictFunc() { 'use strict'; print(typeof this); }
function nonStrictFunc() { print(typeof this); }

strictFunc.call('foo');     // prints 'string' (uncoerced)
nonStrictFunc.call('foo');  // prints 'object' (coerced)
```

Duktape/C functions are currently always **constructable**, i.e. they can always be used in `new Foo()` expressions. You can check whether a function was called in constructor mode as follows:

```
static int my_func(duk_context *ctx) {
    if (duk_is_constructor_call(ctx)) {
        printf("called as a constructor\n");
    } else {
        printf("called as a function\n");
    }
}
```

Storing state for a Duktape/C function

Sometimes it would be nice to provide parameters or additional state to a Duktape/C function out-of-band, i.e. outside explicit call arguments. There are several ways to achieve this.

Function object

First, a Duktape/C function can use its Function object to store state or parameters. A certain Duktape/C function (the actual C function) is always represented by an Ecmascript Function object which is magically associated with the underlying C function. The Function object can be used to store properties related to that particular instance of the function. Note that a certain Duktape/C function can be associated with multiple independent Function objects and thus independent states.

Accessing the Ecmascript Function object related to a Duktape/C function is easy:

```
duk_push_current_function(ctx);
duk_get_prop_string(ctx, -1, "my_state_variable");
```

'this' binding

Another alternative for storing state is to call the Duktape/C function as a method and then use the `this` binding for storing state. For instance, consider a Duktape/C function called as:

```
foo.my_c_func()
```

When called, the Duktape/C function gets `foo` as its `this` binding, and one could store state directly in `foo`. The difference to using the Function object approach is that the same object is shared by all methods, which has both advantages and disadvantages.

Accessing the `this` binding is easy:

```
duk_push_this(ctx);
duk_get_prop_string(ctx, -1, "my_state_variable");
```

Heap stash

The heap stash is an object visible only from C code. It is associated with the Duktape heap, and allows Duktape/C code to store "under the hood" state data which is not exposed to Ecmascript code. It is accessed with the `duk_push_heap_stash()` API call.

Global stash

The global stash is like the heap stash, but is associated with a global object. It is accessed with the `duk_push_global_stash()` API call. There can be several environments with different global objects within the same heap.

Thread stash

The thread stash is like the heap stash, but is associated with a Duktape thread (i.e. a `ctx` pointer). It is accessible with the `duk_push_thread_stash()` API call.

Duktape version specific code

The Duktape version is available through the `DUK_VERSION` define, with the numeric value `(major * 10000) + (minor * 100) + patch`. The same value is available to Ecmascript code through `Duktape.version`. Calling code can use this define for Duktape version specific code.

For C code:

```
#if (DUK_VERSION >= 10203)
/* Duktape 1.2.3 or later */
#elif (DUK_VERSION >= 800)
/* Duktape 0.8.0 or later */
#else
/* Duktape lower than 0.8.0 */
#endif
```

For Ecmascript code (also see Duktape built-ins):

```
if (typeof Duktape !== 'object') {
    print('not Duktape');
} else if (Duktape.version >= 10203) {
    print('Duktape 1.2.3 or higher');
} else if (Duktape.version >= 800) {
    print('Duktape 0.8.0 or higher (but lower than 1.2.3)');
} else {
    print('Duktape lower than 0.8.0');
}
```

Numeric error codes

When errors are created or thrown using the Duktape API, the caller must assign a numeric error code to the error. Error codes are positive integers, with a range restricted to 24 bits at the moment: the allowed error number range is thus [1,16777215]. Built-in error codes are defined in `duktape.h`, e.g. `DUK_ERR_TYPE_ERROR`.

The remaining high bits are used internally to carry e.g. additional flags. Negative error values are used in the Duktape/C API as a shorthand to automatically throw an error.

Error, fatal, and panic

An **ordinary error** is caused by a `throw` statement, a `duk_throw()` API call (or similar), or by an internal, recoverable Duktape error. Ordinary errors can be caught with a `try-catch` in Ecmascript code or e.g. `duk_pcall()` in C code.

An uncaught error or an explicit call to `duk_fatal()` causes a **fatal error** handler to be called. A fatal error handler is associated with every Duktape heap upon creation. There is no reasonable way to resume execution after a fatal error, so the fatal error handler must not return. The default fatal error handler writes an error message to `stderr` and then escalates the fatal error to a panic (which, by default, `abort()`s the process). You can provide your own fatal error handler to deal with fatal errors. The most

appropriate recovery action is, of course, platform and application specific. The handler could, for instance, write a diagnostic file detailing the situation and then restart the application to recover.

A **panic** is caused by Duktape assertion code (if included in the build) or by the default fatal error handler. There is no way to induce a panic from user code. The default panic handler writes an error message to `stderr` and `abort()`s the process. You can use the `DUK_OPT_SEGFAULT_ON_PANIC` feature option to cause a deliberate segfault instead of an `abort()`, which may be useful to get a stack trace from some debugging tools. You can also override the default panic handler entirely with the feature option `DUK_OPT_PANIC_HANDLER`. The panic handler is decided during build, while the fatal error handler is decided at runtime by the calling application.

If assertions are turned off and the application provides a fatal error handler, no panics will be caused by Duktape code. All errors will then be either ordinary errors or fatal errors, both under application control.

Stack types

Duktape stack types are:

Type	Type constant	Type mask constant	Description
(none)	DUK_TYPE_NONE	DUK_TYPE_MASK_NONE	no type (missing value, invalid index, etc)
undefined	DUK_TYPE_UNDEFINED	DUK_TYPE_MASK_UNDEFINED	`undefined`
null	DUK_TYPE_NULL	DUK_TYPE_MASK_NULL	`null`
boolean	DUK_TYPE_BOOLEAN	DUK_TYPE_MASK_BOOLEAN	`true` and `false`
number	DUK_TYPE_NUMBER	DUK_TYPE_MASK_NUMBER	IEEE double
string	DUK_TYPE_STRING	DUK_TYPE_MASK_STRING	immutable string
object	DUK_TYPE_OBJECT	DUK_TYPE_MASK_OBJECT	object with properties
buffer	DUK_TYPE_BUFFER	DUK_TYPE_MASK_BUFFER	mutable byte buffer, fixed/dynamic
pointer	DUK_TYPE_POINTER	DUK_TYPE_MASK_POINTER	opaque pointer (void *)

Memory allocations

The following stack types involve additional heap allocations:

- String: a single allocation contains a combined heap and string header, followed by the immutable string data.
- Object: one allocation is used for a combined heap and object header, and another allocation is used for object properties. The property allocation contains both array entries and normal properties, and if the object is large enough, a hash table to speed up lookups.
- Buffer: for fixed buffers a single allocation contains a combined heap and buffer header, followed by the mutable fixed-size buffer. For dynamic buffers the current buffer is allocated separately.

Note that while strings are considered a primitive (pass-by-value) type in Ecmascript, they are a heap allocated type from a memory allocation viewpoint.

Pointer stability

Heap objects allocated by Duktape have stable pointers: the objects are not relocated in memory while they are reachable from a garbage collection point of view. This is the case for the main heap object, but not necessarily for any additional allocations related to the object, such as dynamic property tables or dynamic buffer data area. A heap object is reachable e.g. when it resides on the value stack of a reachable thread or is reachable through the global object. Once a heap object becomes unreachable any pointers held by user C code referring to the object are unsafe and should no longer be dereferenced.

In practice the only heap allocated data directly referenced by user code are strings, fixed buffers, and dynamic buffers. The data area of strings and fixed buffers is stable; it is safe to keep a C pointer referring to the data even after a Duktape/C function returns as long the string or fixed buffer remains reachable from a garbage collection point of view at all times. Note that this is not usually not the case for Duktape/C value stack arguments, for instance, unless specific arrangements are made.

The data area of a dynamic buffer does **not** have a stable pointer. The buffer itself has a heap header with a stable address but the current buffer is allocated separately and potentially relocated when the buffer is resized. It is thus unsafe to hold a pointer to a dynamic buffer's data area across a buffer resize, and it's probably best not to hold a pointer after a Duktape/C function returns (how would you reliably know when the buffer is resized?).

Type masks

Type masks allows calling code to easily check whether a type belongs to a certain type set. For instance, to check that a certain stack value is a number, string, or an object:

```
if (duk_get_type_mask(ctx, -3) & (DUK_TYPE_MASK_NUMBER |
                                  DUK_TYPE_MASK_STRING |
                                  DUK_TYPE_MASK_OBJECT)) {
    printf("type is number, string, or object\n");
}
```

There is a specific API call for matching a set of types even more conveniently:

```
if (duk_check_type_mask(ctx, -3, DUK_TYPE_MASK_NUMBER |
                                 DUK_TYPE_MASK_STRING |
                                 DUK_TYPE_MASK_OBJECT)) {
    printf("type is number, string, or object\n");
}
```

These are faster and more compact than the alternatives:

```
// alt 1
if (duk_is_number(ctx, -3) || duk_is_string(ctx, -3) || duk_is_object(ctx, -3)) {
    printf("type is number, string, or object\n");
}

// alt 2
int t = duk_get_type(ctx, -3);
if (t == DUK_TYPE_NUMBER || t == DUK_TYPE_STRING || t == DUK_TYPE_OBJECT) {
    printf("type is number, string, or object\n");
}
```

None

The **none** type is not actually a type but is used in the API to indicate that a value does not exist, a stack index is invalid, etc.

Undefined

The **undefined** type maps to Ecmascript `undefined`, which is distinguished from a `null`.

Values read from outside the active value stack range read back as **undefined**.

Null

The **null** type maps to Ecmascript `null`.

Boolean

The **boolean** type is represented in the C API as an integer: zero for false, and non-zero for true.

Whenever giving boolean values as arguments in API calls, any non-zero value is accepted as a "true" value. Whenever API calls return boolean values, the value `1` is always used for a "true" value. This allows certain C idioms to be used. For instance, a bitmask can be built directly based on API call return values, as follows:

```
// this works and generates nice code
int bitmask = (duk_get_boolean(ctx, -3) << 2) |
              (duk_get_boolean(ctx, -2) << 1) |
              duk_get_boolean(ctx, -1);

// more verbose variant not relying on "true" being represented by 1
int bitmask = ((duk_get_boolean(ctx, -3) ? 1 : 0) << 2) |
              ((duk_get_boolean(ctx, -2) ? 1 : 0) << 1) |
              (duk_get_boolean(ctx, -1) ? 1 : 0);

// another verbose variant
int bitmask = (duk_get_boolean(ctx, -3) ? (1 << 2) : 0) |
              (duk_get_boolean(ctx, -2) ? (1 << 1) : 0) |
              (duk_get_boolean(ctx, -1) ? 1 : 0);
```

Number

The **number** type is an IEEE double, including +/- Infinity and NaN values. Zero sign is also preserved. An IEEE double represents all integers up to 53 bits accurately.

IEEE double allows NaN values to have additional signaling bits. Because these bits are used by Duktape internal tagged type representation (when using 8-byte packed values), NaN values in the Duktape API are normalized. Concretely, if you push a certain NaN value to the value stack, another (normalized) NaN value may come out. Don't rely on NaNs preserving their exact form.

String

The **string** type is a raw byte sequence of a certain length which may contain internal NUL (0x00) values. Strings are always automatically NUL terminated for C coding convenience. The NUL terminator is not counted as part of the string length. For instance, the string `"foo"` has byte length 3 and is stored in memory as `{ 'f', 'o', 'o', '\0' }`. Because of the guaranteed NUL termination, strings can always be pointed to using a simple `const char *` as long as internal NULs are not an issue; if they are, the explicit byte length of the string can be queried with the API. Calling code can refer directly to the string data held by Duktape. Such string data pointers are valid (and stable) for as long as a string is reachable in the Duktape heap.

Strings are interned for efficiency: only a single copy of a certain string ever exists at a time. Strings are immutable and must NEVER be changed by calling C code. Doing so will lead to very mysterious issues which are hard to diagnose.

Calling code most often deals with Ecmascript strings, which may contain arbitrary 16-bit codepoints (the whole range 0x0000 to 0xFFFF) but cannot represent non-BMP codepoints (this is how strings are defined in the Ecmascript standard). In Duktape, Ecmascript strings are encoded with CESU-8 encoding. CESU-8 matches UTF-8 except that it allows codepoints in the surrogate pair range (U+D800 to U+DFFF) to be encoded directly; these are prohibited in UTF-8. CESU-8, like UTF-8, encodes all 7-bit ASCII characters as-is which is convenient for C code. For example:

- U+0041 ("A") encodes to `41`.
- U+1234 (ETHIOPIC SYLLABLE SEE) encodes to `e1 88 b4`.
- U+D812 (high surrogate) encodes to `ed a0 92` (this would be invalid UTF-8).

Duktape also allows extended strings internally. Codepoints up to U+10FFFF can be represented with UTF-8, and codepoints above that up to full 32 bits can be represented with "extended UTF-8". Non-standard strings are used for storing internal object properties; using a non-standard string ensures that such properties never conflict with properties accessible using standard Ecmascript strings. Non-standard strings can be given to Ecmascript built-in functions, but since behavior may not be exactly specified, results may vary.

Duktape uses internal object properties to record internal implementation related fields in e.g. function objects. For example, a finalizer reference is stored in an internal finalizer property. Such internal keys are kept separate from valid Ecmascript keys by using a byte sequence which can never occur in a valid CESU-8 string; consequently, standard Ecmascript code cannot accidentally reference such fields. Internal properties are never enumerable, and are not returned by e.g. `Object.getOwnPropertyNames()`. Currently, internal property names begin with an `0xFF` byte followed by the property name. For instance the finalizer property key consists of the byte `0xFF` followed by the ASCII string `"finalizer"`. In internal documentation this property would usually be referred to as `_finalizer` for convenience. You should never read or write internal properties directly.

The "extended UTF-8" encoding used by Duktape is described in the table below. The leading byte is shown in binary (with "x" marking data bits) while continuation bytes are marked with "C" (indicating the bit sequence 10xxxxxx):

Codepoint range	Bits	Byte sequence
U+0000 to U+007F	7	0xxxxxxx
U+0080 to U+07FF	11	110xxxxx C
U+0800 to U+FFFF	16	1110xxxx C C
U+1 0000 to U+1F FFFF	21	11110xxx C C C
U+20 0000 to U+3FF FFFF	26	111110xx C C C C
U+400 0000 to U+7FFF FFFF	31	1111110x C C C C C
U+8000 0000 to U+F FFFF FFFF	36 (32 used)	11111110 C C C C C C

The downside of the encoding for codepoints above U+7FFFFFFF is that the leading byte will be `0xFE` which conflicts with Unicode byte order marker encoding. This is not a practical concern in Duktape's internal use.

Object

The **object** type includes Ecmascript objects and arrays, functions, and threads (coroutines). In other words, anything with properties is an object. Properties are key-value pairs with a string key and an arbitrary value (including **undefined**).

Objects may participate in garbage collection finalization.

Buffer

The **buffer** type is a raw buffer for user data of either fixed or dynamic size. The size of a fixed buffer is given at its creation, and fixed buffers have an unchanging (stable) data pointer. Dynamic buffers may change during their life time at the cost of having a (potentially) changing data pointer. Dynamic buffers also need two memory allocations internally, while fixed buffers only need one. The data pointer of a zero-size dynamic buffer may (or may not) be `NULL` which must be handled by calling code properly (i.e. a `NULL` data pointer only indicates an error if the requested size is non-zero). Unlike strings, buffer data areas are not automatically NUL terminated and calling code must not access the bytes following the allocated buffer size.

Buffers are automatically garbage collected. This also means that C code must not hold onto a buffer data pointer unless the buffer is reachable to Duktape, e.g. resides in an active value stack.

The buffer type is not standard Ecmascript. There are a few different Ecmascript typed array specifications, though, see e.g. Typed

Array Specification. These will be implemented on top of raw arrays, most likely.

Like strings, buffer values have a `length` property and array index properties for reading and writing individual bytes in the buffer. The value of a indexed byte (`buf[123]`) is a number in the range 0...255 which represents a byte value (written values are coerced to integer modulo 256). This differs from string behavior where the indexed values are one-character strings (much more expensive). The `length` property is read-only at the moment (so you can't resize a string by assigning to the length property). These properties are available for both plain buffer values and buffer object values.

A few notes:

- Because the value written to a buffer index is number coerced, assigning a one-character value does not work as often expected. For instance, `buf[123] = 'x'` causes zero to be written to the buffer, as ToNumber('x') = 0. For clarity, you should only assign number values, e.g. `buf[123] = 0x78`.
- There is a fast path for reading and writing numeric indices of plain buffer values, e.g. `x = buf[123]` or `buf[123] = x`. This fast path is not active when the base value is a Buffer object.
- Buffer virtual properties are not currently implemented in `defineProperty()`, so you can't write to buffer indices or buffer `length` with `defineProperty()` now (attempt to do so results in a `TypeError`).

Pointer

The **pointer** type is a raw, uninterpreted C pointer, essentially a `void *`. Pointers can be used to point to native objects (memory allocations, handles, etc), but because Duktape doesn't know their use, they are not automatically garbage collected. You can, however, put one or more pointers inside an object and use the object finalizer to free the native resources related to the pointer(s).

Type algorithms

This section describes how type-related Ecmascript algorithms like comparisons and coercions are extended to Duktape custom types. Duktape specific type algorithms (`ToBuffer()` and `ToPointer()`) are also discussed.

Notation

The following shorthand is used to indicate how values are compared:

Value	Description
t	compares to true
f	compares to false
s	simple compare: boolean-to-boolean, string-to-string (string contents compared), buffer-to-buffer (buffer contents compared), buffer-to-string (buffer and string contents compared)
n	number compare: NaN values compare false, zeroes compare true regardless of sign (e.g. +0 == -0)
N	number compare in SameValue: NaN values compare true, zeroes compare with sign (e.g. SameValue(+0,-0) is false)
p	heap pointer compare
1	string/buffer vs. number: coerce string with ToNumber() and retry comparison; a buffer is first coerced to string and then to number (e.g. buffer with "2.5" coerces eventually to number 2.5)
2	boolean vs. any: coerce boolean with ToNumber() and retry comparison
3	object vs. string/number/buffer: coerce object with ToPrimitive() and retry comparison

Equality (non-strict)

Non-strict equality comparison is specified in The Abstract Equality Comparison Algorithm for standard types. Custom type behavior is as follows:

- Buffer: buffer contents are compared byte-by-byte, buffer values containing the same byte sequence compare equal. This comparison is potentially expensive. (String comparison also compares contents, but because Duktape uses string interning, string content comparison is a cheap pointer compare. This is not the case for buffers.)
- Pointer: comparison against any other type returns false. Comparison to a pointer returns true if and only if the pointer values are the same. Note in particular that comparing a number to a pointer returns false. This seems a bit unintuitive, but numbers cannot represent 64-pointers accurately, comparing numbers and pointers might be error prone.

The standard behavior as well as behavior for Duktape custom types is summarized in the table below:

	und	**nul**	**boo**	**num**	**str**	**obj**	**buf**	**ptr**
und	t	t	f	f	f	f	f	f
nul		t	f	f	f	f	f	f
boo			s	2	2	2	2	f
num				n	1	3	1	f
str					s	3	s	f
obj						p	3	f
buf							s	f
ptr								s

Strict equality

Strict equality is much more straightforward and preferable whenever possible for simplicity and performance. It is described in The Strict Equality Comparison Algorithm for standard types. Custom type behavior is as follows:

- Buffer: compared as heap pointers. Buffer contents are not compared, so this comparison is always fast. Note that this behavior is inconsistent with string comparison behavior: string contents are compared even with strict equality (this is fast, however, due to string interning). This is intentional, as it is important to be able to compare buffer values quickly.
- Pointer: like non-strict equality.

The standard behavior as well as behavior for Duktape custom types is summarized in the table below:

	und	**nul**	**boo**	**num**	**str**	**obj**	**buf**	**ptr**
und	t	f	f	f	f	f	f	f
nul		t	f	f	f	f	f	f
boo			s	f	f	f	f	f
num				n	f	f	f	f
str					s	f	f	f
obj						p	f	f
buf							p	f
ptr								s

SameValue

The `SameValue` algorithm is not easy to invoke from user code. It is used by e.g. `Object.defineProperty()` when checking whether a property value is about to change. SameValue is even stricter than a strict equality comparison, and most notably differs in how numbers are compared. It is specified in The SameValue algorithm for standard types. Custom type behavior is as follows:

- Buffer: like strict equality.
- Pointer: like non-strict (and strict) equality.

The standard behavior as well as behavior for Duktape custom types is summarized in the table below:

	und	nul	boo	num	str	obj	buf	ptr
und	t	f	f	f	f	f	f	f
nul		t	f	f	f	f	f	f
boo			s	f	f	f	f	f
num				N	f	f	f	f
str					s	f	f	f
obj						p	f	f
buf							p	f
ptr								s

Type conversion and testing

The custom types behave as follows for Ecmascript coercions described Type Conversion and Testing (except SameValue which was already covered above):

	buffer	**pointer**
ToPrimitive	identity	identity
ToBoolean	false for zero-size buffer, true otherwise	false for NULL pointer, true otherwise
ToNumber	like ToNumber() for a string	0 for NULL pointer, 1 otherwise
ToInteger	same as ToNumber	same as ToNumber
ToInt32	same as ToNumber	same as ToNumber
ToUint32	same as ToNumber	same as ToNumber
ToUint16	same as ToNumber	same as ToNumber
ToString	string with bytes from buffer data	`sprintf()` with `%p` format (platform specific)
ToObject	Buffer object	Pointer object
CheckObjectCoercible	allow (no error)	allow (no error)
IsCallable	false	false
SameValue	(covered above)	(covered above)

When a buffer is string coerced, the bytes from the buffer are used directly as string data. The bytes will then be interpreted as CESU-8 (or extended UTF-8) from Ecmascript point of view.

Custom coercions (ToBuffer, ToPointer)

ToBuffer() coercion is used when a value is forced into a buffer type e.g. with the `duk_to_buffer()` API call. The coercion is as follows:

- A buffer coerces to itself (identity). The same buffer value is returned.
- Any other type (including pointer) is first string coerced with ToString, and the resulting string is then copied, byte-by-byte, into a fixed-size buffer.

ToPointer() coercion is used e.g. by the `duk_to_pointer()` call. The coercion is as follows:

- A pointer coerces to itself.
- Heap-allocated types (string, object, buffer) coerce to a pointer value pointing to their **internal heap header**. This pointer has only a diagnostic value. Note, in particular, that the pointer returned for a buffer or a string **does not** point to the buffer/string data area. (This coercion is likely to change.)
- Any other types (including number) coerce to a NULL pointer.

The following table summarizes how different types are handled:

	ToBuffer	ToPointer
undefined	buffer with "undefined"	NULL
null	buffer with "null"	NULL
boolean	buffer with "true" or "false"	NULL
number	buffer with string coerced number	NULL
string	buffer with copy of string data	ptr to heap hdr
object	buffer with ToString(value)	ptr to heap hdr
buffer	identity	ptr to heap hdr
pointer	`sprintf()` with `%p` format (platform specific)	identity

Addition

The Ecmascript addition operator is specified in The Addition operator (+). Addition behaves specially if either argument is a string: the other argument is coerced to a string and the strings are then concatenated. This behavior is extended to custom types as follows:

- As for standard types, object values are first coerced with `ToPrimitive()`, e.g. a `Buffer` object is converted to a plain buffer value.
- The string concatenation rule is triggered if either argument is a string or a buffer. The arguments are coerced to strings and then concatenated into the result string. This means that adding two buffers currently results in a string, not a buffer.
- Pointer types fall into the default number addition case. They are coerced with `ToNumber()` and then added as numbers. NULL pointers coerce to 0, non-NULL pointers to 1, so addition results may not be very intuitive.

Property access

If a plain buffer or pointer is used as a property access base value, properties are looked up from the (initial) built-in prototype object (`Duktape.Buffer.prototype` or `Duktape.Pointer.prototype`). This mimics the behavior of standard types.

For example:

```
duk> buf = Duktape.dec('hex', '414243');  // plain buffer
= ABC
duk> buf.toString;
= function toString() {/* native code */}
duk> typeof buf.toString();
= string
```

Duktape built-ins

This section describes Duktape-specific built-in objects, methods, and values.

Additional global object properties

Property	Description
`Duktape`	The Duktape built-in object. Contains miscellaneous implementation specific stuff.
`print`	Non-standard, browser-like function for writing to `stdout`.
`alert`	Non-standard, browser-like function for writing to `stderr`.

print and alert

`print()` writes to `stdout` with an automatic flush afterwards. The bytes written depend on the arguments:

- If given a single buffer argument, the contents of that buffer are written to `stdout` as is. This allows raw byte streams to be reliably written.
- Otherwise arguments are string coerced, joined with a single space character, a newline (0x0a) is appended, and the result is written to `stdout`. For instance, `print('foo', 'bar')` would write the bytes `66 6f 6f 20 62 61 72 0a`. Non-ASCII characters are written directly in their internal extended UTF-8 representation; for most strings this means that output data is properly UTF-8 encoded. Terminal encoding, locale, platform newline conventions etc. have no effect on the output.

`alert()` behaves the same way, but writes to `stderr`. Unlike a browser `alert()`, the call does not block.

The Duktape object

Property	Description
`version`	Duktape version number: `(major * 10000) + (minor * 100) + patch`.
`env`	Cryptic, version dependent summary of most important effective options like endianness and architecture.
`fin`	Set or get finalizer of an object.
`enc`	Encode a value (hex, base-64, JSONX, JSONC): `Duktape.enc('hex', 'foo')`.
`dec`	Decode a value (hex, base-64, JSONX, JSONC): `Duktape.dec('base64', 'Zm9v')`.
`info`	Get internal information (such as heap address and alloc size) of a value in a version specific format.
`line`	Get current line number.
`act`	Get information about call stack entry.
`gc`	Trigger mark-and-sweep garbage collection.
`compact`	Compact the memory allocated for a value (object).
`errcreate`	Callback to modify/replace a created error.
`errthrow`	Callback to modify/replace an error about to be thrown.
`Buffer`	Buffer constructor (function).
`Pointer`	Pointer constructor (function).
`Thread`	Thread constructor (function).
`Logger`	Logger constructor (function).

version

The `version` property allows version-based feature detection and behavior. Version numbers can be compared directly: a logically higher version will also be numerically higher. For example:

```
if (typeof Duktape !== 'object') {
    print('not Duktape');
} else if (Duktape.version >= 10203) {
    print('Duktape 1.2.3 or higher');
} else if (Duktape.version >= 800) {
    print('Duktape 0.8.0 or higher (but lower than 1.2.3)');
} else {
    print('Duktape lower than 0.8.0');
}
```

Unofficial development snapshots have patch level set to 99. For example, version 0.10.99 (1099) would be a development snapshot after 0.10.0 but before the next official release.

Remember to check for existence of `Duktape` when doing feature detection. Your code should typically work on as many engines as possible. Avoid the common pitfall of using a direct identifier reference in the check:

```
// Bad idea: ReferenceError if missing
if (!Duktape) {
    print('not Duktape');
}

// Better: check through 'this' (bound to global)
if (!this.Duktape) {
    print('not Duktape');
}

// Better: use typeof to check also type explicitly
if (typeof Duktape !== 'object') {
    print('not Duktape');
}
```

env

`env` summarizes the most important effective compile options in a version specific, quite cryptic manner. The format is version specific and is not intended to be parsed programmatically. This is mostly useful for developers (see `duk_hthread_builtins.c` for the code which sets the value).

Example from Duktape 0.10.0:

```
ll u p2 a4 x64     // l|b|m integer endianness, l|b|m double endianness,
                   // p|u packed/unpacked tval, p1|p2 prop memory layout,
                   // a1/a4/a8 align target, arch
```

fin()

When called with a single argument, gets the current finalizer of an object:

```
var currFin = Duktape.fin(o);
```

When called with two arguments, sets the finalizer of an object (returns undefined):

```
Duktape.fin(o, function(x) { print('finalizer called'); });
Duktape.fin(o, undefined);  // disable
```

enc()

`enc()` encodes its argument value into chosen format. The first argument is a format (currently supported are "hex", "base64", "jsonx" and "jsonc"), second argument is the value to encode, and any further arguments are format specific.

For "hex" and "base64", buffer values are encoded as is, other values are string coerced and the internal byte representation (extended UTF-8) is then encoded. The result is a string. For example, to encode a string into base64:

```
var result = Duktape.enc('base64', 'foo');
print(result);  // prints 'Zm9v'
```

For "jsonx" and "jsonc" the argument list following the format name is the same as for `JSON.stringify()`: value, replacer (optional), space (optional). For example:

```
var result = Duktape.enc('jsonx', { foo: 123 }, null, 4);
print(result);  // prints JSONX encoded {foo:123} with 4-space indent
```

dec()

`dec()` provides the reverse function of `enc()`.

For "hex" and "base64" the input value is first string coerced (it only really makes sense to decode strings). The result is always a buffer. For example:

```
var result = Duktape.dec('base64', 'Zm9v');
print(typeof result, result);  // prints 'buffer foo'
```

If you wish to get back a string value, you can simply:

```
var result = String(Duktape.dec('base64', 'Zm9v'));
print(typeof result, result);  // prints 'string foo'
```

For "jsonx" and "jsonc" the argument list following the format name is the same as for `JSON.parse()`: text, reviver (optional). For example:

```
var result = Duktape.dec('jsonx', "{foo:123}");
print(result.foo);  // prints 123
```

info()

When given an arbitrary input value, `Duktape.info()` returns an array of values with internal information related to the value. The format of of the values in the array is version specific. This is mainly useful for debugging and diagnosis, e.g. when estimating rough memory usage of objects.

The current result array format is described in the table below. Notes:

- Memory sizes do not include any heap overhead (which may be 8-16 bytes or more, depending on what kind of allocation algorithm is used).
- Reference counts are not adjusted in any way, and include references to the value caused by the `info()` call.
- "type tag" is a number matching `DUK_TYPE_xxx` from `duktape.h`.
- The number of entries allocated for object properties is given by "prop entry count", while "prop entry used" indicates how many of the entries are in used. If an array part is present, "prop array count" indicates the number of entries currently allocated (there is no value to indicate the number of used array part entries). Finally, "prop hash count" indicates the number of entries in a hash lookup table if present (it is not present for typical, small objects). These numbers are counts, not byte sizes.
- Function data contains bytecode instructions, constants, etc. It is shared between all instances (closures) of a certain function template.

Type	0	1	2	3	4	5	6	7	8	9
undefined	type tag	-	-	-	-	-	-	-	-	-
null	type tag	-	-	-	-	-	-	-	-	-
boolean	type tag	-	-	-	-	-	-	-	-	-
number	type tag	-	-	-	-	-	-	-	-	-
string	type tag	heap ptr	refcount	heap hdr size	-	-	-	-	-	-
object, Ecmascript function	type tag	heap ptr	refcount	heap hdr size	prop alloc size	prop entry count	prop entry used	prop array count	prop hash count	func data size
object, Duktape/C function	type tag	heap ptr	refcount	heap hdr size	prop alloc size	prop entry count	prop entry used	prop array count	prop hash count	-
object, thread	type tag	heap ptr	refcount	heap hdr size	prop alloc size	prop entry count	prop entry used	prop array count	prop hash count	-
object, other	type tag	heap ptr	refcount	heap hdr size	prop alloc size	prop entry count	prop entry used	prop array count	prop hash count	-
buffer, fixed	type tag	heap ptr	refcount	heap hdr size	-	-	-	-	-	-
buffer, dynamic	type tag	heap ptr	refcount	heap hdr size	curr buf size	-	-	-	-	-
pointer	type tag	-	-	-	-	-	-	-	-	-

line()

Get the line number of the call site. This may be useful for debugging, e.g.:

```
log.info('reached line:', Duktape.line());
```

NOTE: As `Duktape.act()` provides more information, this function will probably be removed.

act()

Get information about a call stack entry. Takes a single number argument indicating depth in the call stack: -1 is the top entry, -2 is the one below that etc. Returns an object describing the call stack entry, or `undefined` if the entry doesn't exist. Example:

```
function dump() {
    var i, t;
    for (i = -1; ; i--) {
        t = Duktape.act(i);
        if (!t) { break; }
        print(i, t.lineNumber, t.function.name, Duktape.enc('jsonx', t));
    }
}

dump();
```

The example, when executed with the command line tool, currently prints something like:

```
-1 0 act {lineNumber:0,pc:0,function:{_func:true}}
-2 4 dump {lineNumber:4,pc:16,function:{_func:true}}
-3 10 global {lineNumber:10,pc:5,function:{_func:true}}
```

The interesting entries are `lineNumber` and `function` which provides e.g. the function name.

You can also implement a helper to get the current line number using `Duktape.act()`:

```
function getCurrentLine() {
    // indices: -1 = Duktape.act, -2 = getCurrentLine, -3 = caller
    var a = Duktape.act(-3) || {};
    return a.lineNumber;
}
print('running on line:', getCurrentLine());
```

NOTE: The properties provided for call stack entries may change between versions.

gc()

Trigger a forced mark-and-sweep collection. If mark-and-sweep is disabled, this call is a no-op.

compact()

Minimize the memory allocated for a target object. Same as the C API call `duk_compact()` but accessible from Ecmascript code. If called with a non-object argument, this call is a no-op. The argument value is returned by the function, which allows code such as:

```
var obj = {
    foo: Duktape.compact({ bar:123 })
}
```

This call is useful when you know that an object is unlikely to gain new properties, but you don't want to seal or freeze the object in case it does.

errcreate() and errthrow()

These can be set by user code to process/replace errors when they are created (`errcreate`) or thrown (`errthrow`). Both values are initially non-existent.

See Error handlers (errcreate and errthrow) for details.

Duktape.Buffer (constructor)

Property	Description
`prototype`	Prototype for Buffer objects.

The Buffer constructor is a function which returns a plain buffer when called as a normal function and a Buffer object when called as a constructor. Otherwise the behavior is the same:

- If the first argument is a plain buffer, the buffer is used as is (a new buffer is not created). The second argument is ignored. This form can be used to wrap a plain buffer value into a Buffer object.
- If the first argument is a Buffer object, the internal plain buffer value of the argument is used (a new buffer is not created). The second argument is again ignored. If called as a constructor, a new Buffer object is returned, but it will internally point to the same plain buffer value.
- If the first argument is a string, a new buffer is created and the bytes from the string's internal extended UTF-8 representation are copied to the buffer. The second argument (if present) indicates whether or not the new buffer should be dynamic (resizable); the default value is false.
- If the first argument is a number, a new buffer is created and filled with zero. The second argument (if present) indicates whether or not the new buffer should be dynamic. Zero filling the buffer is the default behavior, but this can be disabled with a feature option; if disabled, the contents will be unpredictable.
- When called as a constructor, the result is converted into a Buffer object whose internal value is the plain buffer. The internal prototype of the newly created Buffer will be the `Duktape.Buffer.prototype` object.

NOTE: There is currently (in Duktape 0.10.0) no way direct way to create a copy of a buffer (i.e. a new buffer with the same contents but a separate underlying buffer). This will be added in Duktape 0.11.0; for now you can make a copy inefficiently e.g. as `Duktape.Buffer(String(orig_buf))`. Buffers are currently mostly intended to be operated with from C code.

Duktape.Buffer.prototype

Property	Description
`toString`	Convert Buffer to a printable string.
`valueOf`	Return the primitive buffer value held by Buffer.

`toString()` and `valueOf` accept both plain buffers and Buffer objects as their `this` binding. This allows code such as:

```
var plain_buf = Duktape.Buffer('test');
print(plain_buf.toString());
```

Duktape.Pointer (constructor)

Property	Description
`prototype`	Prototype for Pointer objects.

The Pointer constructor is a function which can be called both as an ordinary function and as a constructor:

- When called as a function, coerces the first argument to a pointer using the custom `ToPointer` coercion. The return value is a plain pointer (not a Pointer object).
- When called as a constructor, coerces the first argument to a pointer using the custom `ToPointer` coercion. Returns a Pointer object whose internal value is the pointer resulting from the coercion. The internal prototype of the newly created Pointer will be the `Duktape.Pointer.prototype` object.

Duktape.Pointer.prototype

Property	Description
`toString`	Convert Pointer to a printable string.
`valueOf`	Return the primitive pointer value held by Pointer.

`toString()` and `valueOf` accept both plain pointers and Pointer objects as their `this` binding. This allows code such as:

```
var plain_ptr = Duktape.Pointer({ test: 'object' });
print(plain_ptr.toString());
```

Duktape.Thread (constructor)

Property	Description
`prototype`	Prototype for Thread objects.
`resume`	Resume target thread with a value or an error. Arguments: target thread, value, flag indicating whether value is to be thrown (optional, default false).
`yield`	Yield a value or an error from current thread. Arguments: value, flag indicating whether value is to be thrown (optional, default false).
`current`	Get currently running Thread object.

The Thread constructor is a function which can be called both as an ordinary function and as a constructor. The behavior is the same in both cases:

- The first argument is checked to be a function (if not, a `TypeError` is thrown). The return value is a new thread whose initial function is recorded to be the argument function (this function will start executing when the new thread is first resumed). The internal prototype of the newly created Thread will be the `Duktape.Thread.prototype` object.

Duktape.Thread.prototype

Property	Description
No properties at the moment.	

Duktape.Logger (constructor)

Property	Description
`prototype`	Prototype for Logger objects.
`clog`	Representative logger for log entries written from C code.

Called as a constructor, creates a new Logger object with a specified name (first argument). If the name is omitted, Logger will

automatically assign a name based on the calling function's `fileName`. If called as a normal function, throws a `TypeError`.

Logger instances have the following properties:

- `n`: logger name; the property will be missing if (a) the given name is not a string, or (b) no name is given and the automatic assignment fails. The logger will then inherit a value from the Logger prototype. You can manually set this property later to whatever value is desired.
- `l`: log level, indicates the minimum log level to output. This property is not assigned by default and the logger inherits a default level from the Logger prototype. You can manually set this property to another value to control log level on a per-logger basis.

To write log entries:

```
var logger = new Duktape.Logger();
logger.info('three values:', val1, val2, val3);
```

For now, see the internal documentation on logging for more info.

Duktape.Logger.prototype

Property	Description
`raw`	Output a formatted log line (buffer value), by default writes to `stderr`.
`fmt`	Format a single (object) argument.
`trace`	Write a trace level (level 0, TRC) log entry.
`debug`	Write a debug level (level 1, DBG) log entry.
`info`	Write an info level (level 2, INF) log entry.
`warn`	Write a warn level (level 3, WRN) log entry.
`error`	Write an error level (level 4, ERR) log entry.
`fatal`	Write a fatal level (level 5, FTL) log entry.
`l`	Default log level, initial value is 2 (info).
`n`	Default logger name, initial value is "anon".

Custom behavior

This section summarizes Duktape behavior which deviates from the E5.1 specification.

Duktape built-in and custom types

The `Duktape` built-in is (of course) non-standard and provides access to Duktape specific features. Also the buffer and pointer types are custom.

Additional Error and Function object properties

See Error objects and Function objects.

Non-strict function instances don't have a `caller` property in the E5/E5.1 specification. Some real world code expects to have this property, so it can be enabled with the feature option `DUK_OPT_FUNC_NONSTD_CALLER_PROPERTY`.

Function statements

E5.1 does not allow a function declaration to appear outside program or function top level:

```
function test() {
    // point A
    try {
        throw new Error('test');
    } catch (e) {
        // This is a SyntaxError in E5.1
        function func() {
            print(typeof e);
        }
        // point B
    }
    // point C
}
```

These declarations are also referred to as "function statements", and appear quite often in real world code (including the test262 test suite), so they are allowed by Duktape. Unfortunately there are several semantics used by different Javascript engines. Duktape follows the V8 behavior for function statements:

- Strict function: a SyntaxError is thrown (standard behavior).
- Non-strict function: treat a function statement like an ordinary function declaration, conceptually "hoisting" it to the top of the function.

As an illustration, the above example would behave as the following:

```
function test() {
    function func() {
        print(typeof e);
    }

    try {
        throw new Error('test');
    } catch (e) {
    }
}
```

`func()` in the above example would already be declared and callable in point A, and would not have access to the `e` binding in any of the points A, B, or C.

RegExp leniency

Although not allowed by E5.1, the following escape is allowed in RegExp syntax:

```
  /\$/       /* matches dollar literally, non-standard */
  /\u0024/   /* same, standard */
```

This escape occurs in real world code so it is allowed. (More leniency will be added in future versions to deal with real world RegExps; dollar escapes are not the only issue.)

Array.prototype.splice() when deleteCount not given

When deleteCount (the 2nd argument) is not given to `Array.prototype.splice()`, the standard behavior is to work as if the 2nd argument was `undefined` (or 0, which has the same behavior after coercions). A more real world compatible behavior is to treat the missing argument like positive infinity, i.e. to extend the splice operation to the end of the array.

Because the non-standard real world behavior is expected by much existing code, Duktape uses this behavior by default. The strict standards compliant behavior can be enabled with the feature option `DUK_OPT_NO_ARRAY_SPLICE_NONSTD_DELCOUNT`.

Custom JSON formats

Ecmascript JSON shortcomings

The standard JSON format has a number of shortcomings when used with Ecmascript:

- `undefined` and function values are not supported
- NaN and infinity values are not supported
- Duktape custom types are, of course, not supported
- Codepoints above BMP cannot be represented except as surrogate pairs
- Codepoints above U+10FFFF cannot be represented even as surrogate pairs
- The output is not printable ASCII which is often inconvenient

These limitations are part of the Ecmascript specification which explicitly prohibits more lenient behavior. Duktape provides two more programmer friendly custom JSON format variants: **JSONX** and **JSONC**, described below.

Custom JSONX format

JSONX encodes all values in a very readable manner and parses back almost all values in a faithful manner (function values being the most important exception). Output is pure printable ASCII, codepoints above U+FFFF are encoded with a custom escape format, and quotes around object keys are omitted in most cases. JSONX is not JSON compatible but a very readable format, most suitable for debugging, logging, etc.

JSONX is used as follows:

```
var obj = { foo: 0/0, bar: [ 1, undefined, 3 ] };
print(Duktape.enc('jsonx', obj));
// prints out: {foo:NaN,bar:[1,undefined,3]}

var dec = Duktape.dec('jsonx', '{ foo: 123, bar: undefined, quux: NaN }');
print(dec.foo, dec.bar, dec.quux);
// prints out: 123 undefined NaN
```

Custom JSONC format

JSONC encodes all values into standard JSON. Values not supported by standard JSON are encoded as objects with a marker key beginning with an underscore (e.g. `{"_ptr":"0xdeadbeef"}`). Such values parse back as ordinary objects. However, you can revive them manually more or less reliably. Output is pure printable ASCII; codepoints above U+FFFF are encoded as plain string data with the format "U+nnnnnnnn" (e.g. `U+0010fedc`).

JSONC is used as follows:

```
var obj = { foo: 0/0, bar: [ 1, undefined, 3 ] };
print(Duktape.enc('jsonc', obj));
// prints out: {"foo":{"_nan":true},"bar":[1,{"_undef":true},3]}

var dec = Duktape.dec('jsonc', '{ "foo": 123, "bar": {"_undef":true}, "quux": {"_nan":tru
print(dec.foo, dec.bar, dec.quux);
// prints out: 123 [object Object] [object Object]
```

The JSONC decoder is essentially the same as the standard JSON decoder at the moment: all JSONC outputs are valid JSON and no custom syntax is needed. As shown in the example, custom values (like `{"_undef":true}`) are **not** revived automatically. They parse back as ordinary objects instead.

Codepoints above U+FFFF and invalid UTF-8 data

All standard Ecmascript strings are valid CESU-8 data internally, so behavior for codepoints above U+FFFF never poses compliance issues. However, Duktape strings may contain extended UTF-8 codepoints and may even contain invalid UTF-8 data.

The Duktape JSON implementation, including the standard Ecmascript JSON API, use replacement characters to deal with invalid UTF-8 data. The resulting string may look a bit odd, but this behavior is preferable to throwing an error.

JSON format examples

The table below summarizes how different values encode in each encoding:

Value	Standard JSON	JSONX	JSONC	Notes
undefined	n/a	`undefined`	`{"_undef":true}`	Standard JSON: encoded as `null` inside arrays, otherwise omitted
null	`null`	`null`	`null`	standard JSON
true	`true`	`true`	`true`	standard JSON
false	`false`	`false`	`false`	standard JSON
123.4	`123.4`	`123.4`	`123.4`	standard JSON
NaN	`null`	`NaN`	`{"_nan":true}`	Standard JSON: always encoded as `null`
Infinity	`null`	`Infinity`	`{"_inf":true}`	Standard JSON: always encoded as `null`
-Infinity	`null`	`-Infinity`	`{"_ninf":true}`	Standard JSON: always encoded as `null`
köhä	`"köhä"`	`"k\xf6h\xe4"`	`"k\u00f6h\u00e4"`	
U+00FC	`"\u00fc"`	`"\xfc"`	`"\u00fc"`	
U+ABCD	`"\uabcd"`	`"\uabcd"`	`"\uabcd"`	
U+1234ABCD	`"U+1234abcd"`	`"\U1234abcd"`	`"U+1234abcd"`	Non-BMP characters are not standard Ecmascript, JSONX format borrowed from Python

object	`{"my_key":123}`	`{my_key:123}`	`{"my_key":123}`	ASCII keys matching identifer requirements encoded without quotes in JSONX
array	`["foo","bar"]`	`["foo","bar"]`	`["foo","bar"]`	
buffer	`n/a`	`|deadbeef|`	`{"_buf":"deadbeef"}`	
pointer	`n/a`	`(0xdeadbeef)` `(DEADBEEF)`	`{"_ptr":"0xdeadbeef"}` `{"_ptr":"DEADBEEF"}`	Representation inside parentheses or quotes is platform specific
NULL pointer	`n/a`	`(null)`	`{"_ptr":"null"}`	
function	`n/a`	`{_func:true}`	`{"_func":true}`	Standard JSON: encoded as `null` inside arrays, otherwise omitted

Limitations

Some limitations include:

- Only enumerable own properties are serialized in any of the formats.
- Array properties (other than the entries) are not serialized. This would be useful in e.g. logging, e.g. as `[1,2,3,"type":"point"]`.
- There is no automatic revival of special values when parsing JSONC data.
- There is no canonical encoding. This would be easy to arrange with a simple option to sort object keys during encoding.

(See internal documentation for more future work issues.)

Error objects

Property summary

Ecmascript Error objects have very few standard properties, so many Ecmascript implementations have added quite a few custom properties. Duktape uses standard Error properties but also borrows the most useful properties used by other implementations. The number of "own" properties of error objects is minimized to keep error objects as small as possible.

Error objects have the following properties (mostly inherited):

Property name	Compatibility	Description
`name`	standard	Name of error, e.g. `TypeError`, inherited
`message`	standard	Optional message of error, own property, empty message inherited if absent
`fileName`	Rhino	Filename related to error source, inherited accessor
`lineNumber`	Rhino	Linenumber related to error source, inherited accessor
`stack`	V8	Traceback as a multi-line human redable string, inherited accessor
`tracedata`	Duktape	Raw traceback data in an internal format, own property

If Duktape is compiled with traceback support:

- `stack`, `fileName`, and `lineNumber` are accessor properties inherited from `Error.prototype`.
- `tracedata` is an own property of the Error instance which provides the raw data (in an internal format) needed by the

accessors.

If Duktape is compiled without traceback support:

- The `stack` accessor will be equivalent to `Error.prototype.toString()`, so that printing the stacktrace always produces a useful result.
- `fileName` and `lineNumber` will be own properties of the Error object.

When error objects are created using the Duktape API from C code and the caller does not give a format string for a `message`, the `message` property is set to a numeric error code given in the API call. The type of `message` will be number in this case; normally error messages are strings. In minimized Duktape builds all errors generated internally by Duktape use numeric error codes only.

An object is considered an "error object" if its internal prototype chain contains the (original) `Error.prototype` object. Only objects matching this criteria get augmented with e.g. traceback data.

Traceback

The `stack` property is an accessor (setter/getter) property which provides a printable traceback related to an error. The traceback reflects the call stack when the error object was created (not thrown). Traceback data is automatically collected and added to an object:

- when an Error instance is constructed;
- when an error is thrown from C code using the Duktape API;
- when an error is thrown from inside Duktape.

The data used to create the traceback is stored in the `tracedata` property in an internal and version-dependent format described in the internal documentation. You shouldn't access the `tracedata` directly.

The exact traceback format is still in flux (and you shouldn't rely on an exact format in any case). As an example, the program:

```
// shortened from ecmascript-testcases/test-dev-traceback-example.js
try {
    decodeURIComponent('%e1%a9%01');  // invalid utf-8
} catch (e) {
    print(e.stack);
}
```

would print something like:

```
URIError: invalid input
        duk_bi_global.c:317
        decodeURIComponent (null) native strict
        global ecmascript-testcases/test-dev-traceback-example.js:3
```

In builds where tracebacks are disabled, the `stack` accessor will return the same value as calling `toString()` on the error would. This means you can always print `e.stack` and get a useful output.

The most portable traceback printing approach is something like:

```
try {
    decodeURIComponent('%e1%a9%01');  // invalid utf-8
} catch (e) {
    // Print stacktrace on at least Duktape and V8, or a standard error
    // string otherwise.
    print(e.stack || e);
}
```

Attempt to write to `stack` is silently ignored. You can still override the accessor by defining an own property of the same name explicitly with `Object.defineProperty()`. This behavior differs from V8 where `stack` is an own property of the Error instance, and if you assign a value to `stack`, the value reads back as assigned.

Error handlers (errcreate and errthrow)

If `Duktape.errcreate` has been set, it is called right after Duktape has added traceback information to an object, and can process the error further or even replace the error value entirely. The error handler only gets called with `Error` instances, and its return value is used as the final error value. If the error handler throws an error, that error replaces the original error. The error handler is usually called only once per error. However, in corner cases related to constructors, the error handler can be called multiple times for a single error value.

An error handler should avoid overwriting any properties already present in an object, as that would be quite confusing for other code. In general, an error handler should always avoid throwing an error, as that error replaces the original error and would also be confusing. As a specific example, an error handler must not try to add a new property to a non-extensible object, as that would cause a `TypeError`.

Below is an example error handler for adding a creation timestamp to errors at their creation:

```
Duktape.errcreate = function (e) {
    if (!(e instanceof Error)) {
        // this check is not really needed because errcreate only gets
        // called with Error instances
        return e;
    }
    if ('created' in e) {
        // already augmented or conflicting property present
        return e;
    }
    if (!Object.isExtensible(e)) {
        // object not extensible, don't try to add a new property
        return e;
    }
    e.created = new Date();
    return e;
}
```

To remove the handler, delete the property (setting it to e.g. `null` does not work and causes a `TypeError` when Duktape attempts to call the `null` value):

```
// Remove error handler for error creation
delete Duktape.errcreate;
```

Similarly, if `Duktape.errthrow` has been set, it is called right before an error is thrown, and can process or replace the error value. Because Ecmascript allows any value type to be thrown, the error handler may get called with arbitrary input values (not just `Error` instances). It may also be called more than once for the same value because an error can be re-thrown multiple times.

For example, to add a throw timestamp (recording the first time the object has been thrown) to errors:

```
Duktape.errcreate = function (e) {
    if (!(e instanceof Error)) {
        // refuse to touch anything but Error instances
        return e;
    }
    if ('thrown' in e) {
        // already augmented or conflicting property present
        return e;
    }
    if (!Object.isExtensible(e)) {
        // object not extensible, don't try to add a new property
        return e;
    }
    e.thrown = new Date();
    return e;
}
```

Again, to remove the handler, delete the property:

```
// Remove error handler for error throwing
delete Duktape.errthrow;
```

Current limitations

- There is no cause chain support. Cause chains would be useful but there are no cause chains in Ecmascript, nor does there seem to be a de facto standard for them.
- These is currently no way to access traceback elements programmatically.
- If an error is created with a non-constructor function call to a custom error class (`MyError('msg')` instead of `new MyError('msg')`) it won't get augmented with custom fields such as traceback data. When called as a constructor custom errors inheriting from `Error` get augmented normally. Built-in standard errors (like `TypeError`) always get augmented, even when created with a non-constructor function call (the tracebacks look slightly different depending on how the error is created, though).

Function objects

Property summary

Duktape Function objects add a few properties to standard Ecmascript properties. The table below summarizes properties assigned to

newly created function instances (properties can of course be added or removed afterwards):

Property name	Compatibility	Description
`length`	standard	Function argument count (if relevant). Present for all Function objects, including bound functions.
`prototype`	standard	Prototype used for new objects when called as a constructor. Present for most constructable Function objects, not copied to bound functions.
`caller`	standard	Accessor which throws an error. Present for strict functions and bound functions. Not copied to bound functions. (If `DUK_OPT_FUNC_NONSTD_CALLER_PROPERTY` is given, non-strict functions will get a non-standard `caller` property.)
`arguments`	standard	Accessor which throws an error. Present for strict functions and bound functions. Not copied to bound functions.
`name`	Duktape	Function name, see below. Copied to bound function from target function.
`fileName`	Duktape	Filename or context where function was declared (same name as in error tracebacks). Copied to bound function from target function.
`callee`	n/a	Never assigned by default (listed here to clarify relationship to "caller" property).

The `name` property is assigned to all functions and is also the name used in tracebacks. It is assigned as follows:

```
function funcDecl() {
    /* Function declaration: 'name' is declaration name, here 'funcDecl'. */
}

var foo = function namedFunc() {
    /* Named function expression: 'name' is the name used in expression,
     * here 'namedFunc' (not 'foo').
     */
}

var bar = function () {
    /* Anonymous function expression: 'name' is the empty string. */
}
```

User-created Duktape/C functions (`duk_push_c_function()`) have a different set of properties to reduce Function object memory footprint:

Property name	Compatibility	Description
`length`	standard	Function argument count, matches argument to `duk_push_c_function()`, 0 for varargs. Non-writable and non-configurable.

Note in particular that the standard `prototype`, `caller`, and `arguments` properties are missing by default. This is not strictly compliant but is important to reduce function footprint. User code can of course assign these but is not required to do so.

Finalization

Overview of finalization

An object which has an internal finalizer property in its prototype chain (or in the object itself) is subject to finalization before being freed. The internal finalizer property is set using `Duktape.fin()` method with the object and the finalizer function as call arguments. The current finalizer is read back by calling `Duktape.fin()` with only the object as a call argument.

The finalizer method is called with the target object as its sole argument. The method may rescue the object by creating a live reference to the object before returning. The return value is ignored; similarly, any errors thrown by the finalizer are ignored. The finalizer may be triggered by either reference counting or mark-and-sweep.

Finalizers cannot currently yield. The context executing the finalization can currently be any coroutine in the heap. (This will be fixed in the future.)

Example

Finalization example:

```
// finalize.js
var a;

function init() {
    a = { foo: 123 };

    Duktape.fin(a, function (x) {
        print('finalizer, foo ->', x.foo);
    });
}

// create object, reference it through 'a'
init();

// delete reference, refcount triggers finalization immediately
print('refcount finalizer');
a = null;

// mark-and-sweep finalizing happens here (at the latest) if
// refcounting is disabled
print('mark-and-sweep finalizer')
Duktape.gc();
```

If you run this with the Duktape command line tool (with the default Duktape profile), you'll get:

```
$ duk finalize.js
refcount finalizer
finalizer, foo -> 123
mark-and-sweep finalizer
Cleaning up...
```

Coroutines

Overview of coroutines

Duktape has a support for simple coroutines. Execution is strictly nesting: coroutine A resumes or initiates coroutine B, coroutine B runs until it yields or finishes (either successfully or through an uncaught error), after which coroutine A continues execution with the yield result.

Coroutines are created with `new Duktape.Thread()`, which gets as its sole argument the initial function where the new coroutine begins execution on its first resume. The resume argument becomes the initial function's first (and only) argument value.

A coroutine is resumed using `Duktape.Thread.resume()` which takes the following arguments: the coroutine to resume, the resume value, and (optionally) a flag indicating whether the resume value is an ordinary value or an error to be injected into the target coroutine. Injecting an error means that the resume value will be "thrown" at the site of the target coroutine's last yield operation. In other words, instead of returning with an ordinary value, the yield will seemingly throw an error.

A coroutine yields its current execution using `Duktape.Thread.yield()` which takes as its arguments: the value to yield, and (optionally) a flag indicating whether the yield value is an ordinary value or an error to be thrown in the context of the resuming coroutine. In other words, an error value causes the resume operation to seemingly throw an error instead of returning an ordinary value.

If a coroutine exists successfully, i.e. the initial function finishes by returning a value, it is handled similarly to a yield with the return value. If a coroutine exists because of an uncaught error, it is handled similarly to a yield with the error: the resume operation will rethrow that error in the resuming coroutine's context. In either case the coroutine which has finished can no longer be resumed; attempt to do so will cause a TypeError.

There are currently strict limitations on when a yield is possible. In short, a coroutine can only yield if its entire active call stack consists of plain Ecmascript-to-Ecmascript calls. The following prevent a yield if they are present anywhere in the yielding coroutine's call stack:

- a Duktape/C function call
- a constructor call
- a getter/setter call
- an `eval()` call
- `Function.prototype.call()` or `Function.prototype.apply()`
- a finalizer call

Example

A simple example of the basic mechanics of spawning, resuming, and yielding:

```
// coroutine.js
function yielder(x) {
    var yield = Duktape.Thread.yield;

    print('yielder starting');
    print('yielder arg:', x);

    print('resumed with', yield(1));
    print('resumed with', yield(2));
    print('resumed with', yield(3));

    print('yielder ending');
    return 123;
}

var t = new Duktape.Thread(yielder);

print('resume test');
print('yielded with', Duktape.Thread.resume(t, 'foo'));
print('yielded with', Duktape.Thread.resume(t, 'bar'));
print('yielded with', Duktape.Thread.resume(t, 'quux'));
print('yielded with', Duktape.Thread.resume(t, 'baz'));
print('finished');
```

When executed with the duk command line tool, this prints:

```
$ duk coroutine.js
resume test
yielder starting
yielder arg: foo
yielded with 1
resumed with bar
yielded with 2
resumed with quux
yielded with 3
resumed with baz
yielder ending
yielded with 123
finished
```

Compiling

Automatic defaults

If you compile Duktape with no compiler options, Duktape will detect the compiler and the platform automatically and select defaults appropriate in most cases. Recommended compiler options (for GCC/clang, use similar options in your compiler):

- `-std=c99`: recommended to ensure C99 semantics which improve C type detection and allows Duktape to use variadic macros (without these you may get harmless compiler warnings because of the way Duktape works around the lack of variadic macros)
- `-Os`: optimize for smallest footprint, which is usually desired when embedding Duktape
- `-fomit-frame-pointer`: omit frame pointer, further reduces footprint but may interfere with debugging (leave out from debug builds)
- `-fstrict-aliasing`: use strict aliasing rules, Duktape is compatible with these and they improve the resulting C code

Duktape feature defaults are, at a high level:

- Full Ecmascript compliance (including the optional Annex B features), except for intentional real world compatibility deviations (see Custom behavior)
- Packed value representation (8 bytes per value) when available, unpacked value representation (12-16 bytes per value) when not
- Reference counting and mark-and-sweep garbage collection
- Full error messages and tracebacks
- No debug printing, no asserts, etc

Usually the automatic defaults are OK. If you're using Duktape on a platform where Duktape's automatic feature detection doesn't (yet) work, you may need to force a specific byte order or alignment requirements with **feature options** described below.

Feature options (DUK_OPT_xxx)

If you wish to modify the defaults, you can provide feature options in the form of `DUK_OPT_xxx` compiler defines. These will be taken into account by the internal `duk_features.h` file, which resolves the final internal features based on feature requests, compiler features, and platform features.

The available feature options can be found in `duk_features.h`. The table below summarizes the available options, in no particular order:

Define	Description
`DUK_OPT_NO_PACKED_TVAL`	Don't use the packed 8-byte internal value representation even if otherwise possible. The packed representation has more platform/compiler portability issues than the unpacked one.
`DUK_OPT_FORCE_ALIGN`	Use `-DDUK_OPT_FORCE_ALIGN=4` or `-DDUK_OPT_FORCE_ALIGN=8` to force a specific struct/value alignment instead of relying on Duktape's automatic detection. This shouldn't normally be needed.
`DUK_OPT_FORCE_BYTEORDER`	Use this to skip byte order detection and force a specific byte order: `1` for little endian, `2` for ARM "mixed" endian (integers little endian, IEEE doubles mixed endian), `3` for big endian. Byte order detection relies on unstandardized platform specific header files, so this may be required for custom platforms if compilation fails in endianness detection.
`DUK_OPT_DEEP_C_STACK`	By default Duktape imposes a sanity limit on the depth of the C stack because it is often limited in embedded environments. This option forces Duktape to use a deep C stack which relaxes e.g. recursion limits. Automatic feature detection enables deep C stacks on some platforms known to have them (e.g. Linux, BSD, Windows).
`DUK_OPT_NO_REFERENCE_COUNTING`	Disable reference counting and use only mark-and-sweep for garbage collection. Although this reduces memory footprint of heap objects, the downside is much more fluctuation in memory usage.
`DUK_OPT_NO_MARK_AND_SWEEP`	Disable mark-and-sweep and use only reference counting for garbage collection. This reduces code footprint and eliminates

	garbage collection pauses, but objects participating in unreachable reference cycles won't be collected until the Duktape heap is destroyed. In particular, function instances won't be collected because they're always in a reference cycle with their default prototype object. Unreachable objects are collected if you break reference cycles manually (and are always freed when a heap is destroyed).
DUK_OPT_NO_VOLUNTARY_GC	Disable voluntary periodic mark-and-sweep collection. A mark-and-sweep collection is still triggered in an out-of-memory condition. This option should usually be combined with reference counting, which collects all non-cyclical garbage. Application code should also request an explicit garbage collection from time to time when appropriate. When this option is used, Duktape will have no garbage collection pauses in ordinary use, which is useful for timing sensitive applications like games.
DUK_OPT_NO_MS_STRINGTABLE_RESIZE	Disable forced string intern table resize during mark-and-sweep garbage collection. This may be useful when reference counting is disabled, as mark-and-sweep collections will be more frequent and thus more expensive.
DUK_OPT_GC_TORTURE	Development time option: force full mark-and-sweep on every allocation to stress test memory management.
DUK_OPT_NO_AUGMENT_ERRORS	Don't augment Ecmascript error objects with custom fields like `fileName`, `lineNumber`, and traceback data. Also disables `Duktape.errcreate` and `Duktape.errthrow` error handler callbacks. Implies DUK_OPT_NO_TRACEBACKS.
DUK_OPT_NO_TRACEBACKS	Don't record traceback data into Ecmascript error objects (but still record `fileName` and `lineNumber`). Reduces footprint and makes error handling a bit faster, at the cost of less nformative Ecmascript errors.
DUK_OPT_NO_VERBOSE_ERRORS	Don't provide error message strings or filename/line information for errors generated by Duktape. Reduces footprint, at the cost of much less informative Ecmascript errors.
DUK_OPT_TRACEBACK_DEPTH	Override default traceback collection depth. The default is currently 10.
DUK_OPT_NO_PC2LINE	Don't record a "pc2line" map into function instances. Without this map, exceptions won't have meaningful line numbers (virtual machine program counter values cannot be translated to line numbers) but function instances will have a smaller footprint.
DUK_OPT_NO_REGEXP_SUPPORT	Disable support for regular expressions. Regexp literals are treated as a `SyntaxError`, RegExp constructor and prototype functions throw an error, `String.prototype.replace()` throws an error if given a regexp search value, `String.prototype.split()` throws an error if given a regexp separator value, `String.prototype.search()` and `String.prototype.match()` throw an error unconditionally.
DUK_OPT_STRICT_UTF8_SOURCE	Enable strict UTF-8 parsing of source code. When enabled, non-shortest encodings (normally invalid UTF-8) and surrogate pair codepoints are accepted as valid source code characters. This option breaks compatibility with some test262 tests.
DUK_OPT_NO_OCTAL_SUPPORT	Disable optional octal number support (Ecmascript E5/E5.1 Annex B).
DUK_OPT_NO_SOURCE_NONBMP	Disable accurate Unicode support for non-BMP characters in

	source code. Non-BMP characters are then always accepted as identifier characters.
`DUK_OPT_NO_BROWSER_LIKE`	Disable browser-like functions. Makes `print()` and `alert()` throw an error. This option is confusing when used with the Duktape command line tool, as the command like tool will immediately panic.
`DUK_OPT_NO_SECTION_B`	Disable optional features in Ecmascript specification Annex B. Causes `escape()`, `unescape()`, and `String.prototype.substr()` to throw an error.
`DUK_OPT_NO_FUNC_STMT`	Disable support for function declarations outside program or function top level (also known as "function statements"). Such declarations are non-standard and the strictly compliant behavior is to treat them as a SyntaxError. Default behavior is to treat them like ordinary function declarations ("hoist" them to function top) with V8-like semantics.
`DUK_OPT_FUNC_NONSTD_CALLER_PROPERTY`	Add a non-standard `caller` property to non-strict function instances for better compatibility with existing code. The semantics of this property are not standardized and may vary between engines; Duktape tries to behave close to V8 and Spidermonkey. See Mozilla description of the property. This feature disables tail call support.
`DUK_OPT_FUNC_NONSTD_SOURCE_PROPERTY`	Add a non-standard `source` property to function instances. This allows function `toString()` to print out the actual function source. The property is disabled by default because it increases memory footprint.
`DUK_OPT_NO_ARRAY_SPLICE_NONSTD_DELCOUNT`	For better compatibility with existing code, `Array.prototype.splice()` has non-standard behavior by default when the second argument (deleteCount) is not given: the splice operation is extended to the end of the array. If this option is given, `splice()` will behave in a strictly conforming fashion, treating a missing deleteCount the same as an undefined (or 0) value.
`DUK_OPT_NO_JSONX`	Disable support for the JSONX format. Reduces code footprint. Causes JSONX calls to throw an error.
`DUK_OPT_NO_JSONC`	Disable support for the JSONC format. Reduces code footprint. Causes JSONC calls to throw an error.
`DUK_OPT_NO_FILE_IO`	Disable use of ANSI C file I/O which might be a portability issue on some platforms. Causes `duk_eval_file()` to throw an error, makes built-in `print()` and `alert()` no-ops, and suppresses writing of a panic message to `stderr` on panic. This option does not suppress debug printing so don't enable debug printing if you wish to avoid I/O.
`DUK_OPT_NO_INTERRUPT_COUNTER`	Disable the internal bytecode executor periodic interrupt counter. The mechanism is used to implement e.g. execution step limit, custom profiling, and debugger interaction. Disabling the interrupt counter improves bytecode execution performance very slightly but disables all features depending on it.
`DUK_OPT_NO_ZERO_BUFFER_DATA`	By default Duktape zeroes data allocated for buffer values. Define this to disable the zeroing (perhaps for performance reasons).
`DUK_OPT_PANIC_HANDLER(code,msg)`	Provide a custom panic handler, see detailed description below.
`DUK_OPT_DECLARE`	Provide declarations or additional #include directives to be used when compiling Duktape. You may need this if you set `DUK_OPT_PANIC_HANDLER` to call your own panic handler function (see example below). You can also use this option to

	cause additional files to be included when compiling Duktape.
DUK_OPT_SEGFAULT_ON_PANIC	Cause the default panic handler to cause a segfault instead of using `abort()` or `exit()`. This is useful when debugging with valgrind, as a segfault provides a nice C traceback in valgrind.
DUK_OPT_SELF_TESTS	Perform run-time self tests when a Duktape heap is created. Catches platform/compiler problems which cannot be reliably detected during compile time. Not enabled by default because of the extra footprint.
DUK_OPT_ASSERTIONS	Enable internal assert checks. These slow down execution considerably so only use when debugging.
DUK_OPT_DEBUG	Enable debug printouts.
DUK_OPT_DDEBUG	Enable more debug printouts.
DUK_OPT_DDDEBUG	Enable even more debug printouts. Not recommended unless you have grep handy.
DUK_OPT_DPRINT_COLORS	Enable coloring of debug prints with ANSI escape codes. The behavior is not sensitive to terminal settings.
DUK_OPT_DPRINT_RDTSC	Print RDTSC cycle count in debug prints if available.
DUK_OPT_DEBUG_BUFSIZE	Debug code uses a static buffer as a formatting temporary to avoid side effects in debug prints. The static buffer is large by default, which may be an issue in constrained environments. You can set the buffer size manually with this option. Example: `-DDUK_OPT_DEBUG_BUFSIZE=2048`.
DUK_OPT_HAVE_CUSTOM_H	Enable user-provided `duk_custom.h` customization header (see below for details). Not recommended unless really necessary.

Suggested feature options for some environments

Timing sensitive applications (e.g. games)

- Use the default memory management settings (reference counting and mark-and-sweep) but enable `DUK_OPT_NO_VOLUNTARY_GC` to eliminate mark-and-sweep pauses. Use explicit GC calls (either `duk_gc()` from C or `Duktape.gc()` from Ecmascript) when possible to collect circular references.

Memory constrained applications

- Use the default memory management settings: although reference counting increases heap header size, it also reduces memory usage fluctuation which is often more important than absolute footprint.
- Reduce error handling footprint with one or more of: `DUK_OPT_NO_AUGMENT_ERRORS`, `DUK_OPT_NO_TRACEBACKS`, `DUK_OPT_NO_VERBOSE_ERRORS`, `DUK_OPT_NO_PC2LINE`.
- If you don't need the Duktape-specific additional JSONX/JSONC formats, use both `DUK_OPT_NO_JSONX` and `DUK_OPT_NO_JSONC`.
- If you don't need regexp support, use `DUK_OPT_NO_REGEXP_SUPPORT`.
- Duktape debug code uses a large, static temporary buffer for formatting debug log lines. Use e.g. `-DDUK_OPT_DEBUG_BUFSIZE=2048` to reduce this overhead.

DUK_OPT_HAVE_CUSTOM_H and duk_custom.h

Normally you define `DUK_OPT_xxx` feature options and the internal `duk_features.h` header resolves these with platform/compiler constraints to determine effective compilation options for Duktape internals. The effective options are provided as `DUK_USE_xxx` defines which you normally never see.

If you define DUK_OPT_HAVE_CUSTOM_H, Duktape will include duk_custom.h after determining the appropriate DUK_USE_xxx defines but before compiling any code. The duk_custom.h header, which you provide, can then tweak the active DUK_USE_xxx defines freely. See duk_features.h for the available defines.

This approach is useful when the DUK_OPT_xxx feature options don't provide enough flexibility to tweak the build. The downside is that you can easily create inconsistent DUK_USE_xxx flags, the customization header will be version specific, and you need to peek into Duktape internals to know what defines to tweak.

DUK_OPT_PANIC_HANDLER

The default panic handler will print an error message to stdout unless I/O is disabled by DUK_OPT_NO_FILE_IO. It will then call abort() or cause a segfault if DUK_OPT_SEGFAULT_ON_PANIC is defined.

You can override the entire panic handler by defining DUK_OPT_PANIC_HANDLER. For example, you could add the following to your compiler options:

```
'-DDUK_OPT_PANIC_HANDLER(code,msg)={printf("*** %d:%s\n",(code),(msg));abort();}'
```

You can also use:

```
'-DDUK_OPT_PANIC_HANDLER(code,msg)={my_panic_handler((code),(msg))}'
```

which calls your custom handler:

```
void my_panic_handler(int code, const char *msg) {
    /* Your panic handling.  Must not return. */
}
```

The DUK_OPT_PANIC_HANDLER macro is used internally by Duktape, so your panic handler function needs to be declared for Duktape compilation to avoid compiler warnings about undeclared functions. You can "inject" a declaration for your function into Duktape compilation with:

```
'-DDUK_OPT_DECLARE=extern void my_panic_handler(int code, const char *msg);'
```

After this you might still get a compilation warning like "a noreturn function must not return" as the compiler doesn't know your panic handler doesn't return. You can fix this by either using a (compiler specific) "noreturn" declaration, or by modifying the panic handler macro to something like:

```
'-DDUK_OPT_PANIC_HANDLER(code,msg)={my_panic_handler((code),(msg));abort()}'
```

As abort() is automatically a "noreturn" function the panic macro body can no longer return. Duktape always includes stdlib.h which provides the abort() prototype so no additional include files are needed.

Adding new feature options

This section only applies if you customize Duktape internals and wish to submit a patch to be included in the mainline distribution:

- Add a descriptive DUK_OPT_xxx for the custom feature. The custom feature should only be enabled if the feature option is explicitly given.
- Modify duk_features.h to detect your custom feature option and define appropriate internal DUK_USE_xxx define(s).

Conflicts with other features should be detected. Code outside `duk_features.h` should only listen to `DUK_USE_xxx` defines so that the resolution process is fully contained in `duk_features.h`.

Memory management alternatives

There are three supported memory management alternatives:

- **Reference counting and mark-and-sweep (default)**: heap objects are freed immediately when they become unreachable except for objects participating in unreachable reference cycles. Such objects are freed by a periodic voluntary, stop the world mark-and-sweep collection. Mark-and-sweep is also used as the emergency garbage collector if memory allocation fails.
- **Reference counting only**: reduces code footprint and eliminates garbage collection pauses, but objects in unreachable reference cycles are not collected until the Duktape heap is destroyed. This alternative is not recommended unless the reference cycles are not an issue. See notes below.
- **Mark-and-sweep only**: reduces code footprint and memory footprint (heap headers don't need to store a reference count), but there is more memory usage variance than in the default case. The frequency of voluntary, stop the world mark-and-sweep collections is also higher than in the default case where reference counting is expected to handle almost all memory management.

When using only reference counting it is important to avoid creating unreachable reference cycles. Reference cycles are usually easy to avoid in application code e.g. by using only forward pointers in data structures. Even if reference cycles are necessary, garbage collection can be allowed to work simply by breaking the cycles before deleting the final references to such objects. For example, if you have a tree structure where nodes maintain references to both children and parents (creating reference cycles for each node) you could walk the tree and set the parent reference to `null` before deleting the final reference to the tree.

Unfortunately every Ecmascript function instance is required to be in a reference loop with an automatic prototype object created for the function. You can break this loop manually if you wish. For internal technical reasons, named function expressions are also in a reference loop; this loop cannot be broken from user code and only mark-and-sweep can collect such functions. See Limitations.

Compiler warnings

Current goal is for the Duktape compile to be clean when:

- using a major compiler (e.g. gcc, clang, MSVC, mingw);
- the compiler is in C99 mode; and
- warnings are enabled (e.g. `-Wall` in gcc/clang).

There are still some warnings present when you compile with `-Wextra` or equivalent option.

When your compiler is not C99 compliant, Duktape uses an awkward replacement for variadic macros. This may cause, as a side effect, a lot of harmless warnings if you set the compiler warning level too high. This is difficult to fix, so C99 compilation may not be clean at the moment.

Performance

This section discussed Duktape specific performance characteristics and provides some hints to avoid Duktape specific performance pitfalls.

Duktape performance characteristics

String interning

Strings are interned: only a single copy of a certain string exists at any point in time. Interning a string involves hashing the string and looking up a global string table to see whether the string is already present. If so, a pointer to the existing string is returned; if not, the

string is inserted into the string table, potentially involving a string table resize. While a string remains reachable, it has a unique and a stable pointer which allows byte-by-byte string comparisons to be converted to simple pointer comparisons. Also, string hashes are computed during interning which makes the use of string keys in internal hash tables efficient.

There are many downsides also. Strings cannot be modified in-place but a copy needs to be made for every modification. For instance, repeated string concatenation creates a temporary value for each intermediate string which is especially bad if a result string is built one character at a time. Duktape internal primitives, such as string case conversion and array `join()`, try to avoid these downsides by minimizing the number of temporary strings created.

String memory representation and the string cache

The internal memory representation for strings is extended UTF-8, which represents each ASCII character with a single byte but uses two or more bytes to represent non-ASCII characters. This reduces memory footprint for most strings and makes strings easy to interact with in C code. However, it makes random access expensive for non-ASCII strings. Random access is needed for operations such as extracting a substring or looking up a character at a certain character index.

Duktape automatically detects pure ASCII strings (based on the fact that their character and byte length are identical) and provides efficient random access to such strings.

However, when a string contains non-ASCII characters a **string cache** is used to resolve a character index to an internal byte index. Duktape maintains a few (internal define `DUK_HEAP_STRCACHE_SIZE`, currently 4) string cache entries which remember the last byte offset and character offset for recently accessed strings. Character index lookups near a cached character/byte offset can be efficiently handled by scanning backwards or forwards from the cached location. When a string access cannot be resolved using the cache, the string is scanned either from the beginning or the end, which is obviously very expensive for large strings. The cache is maintained with a very simple LRU mechanism and is transparent to both Ecmascript and C code.

The string cache makes simple loops like the following efficient:

```
var i;
var n = inp.length;
for (i = 0; i < n; i++) {
    print(inp.charCodeAt(i));
}
```

When random accesses are made from here and there to multiple strings, the strings may very easily fall out of the cache and become expensive at least for longer strings.

Note that the cache never maintains more than one entry for each string, so the following would be very inefficient:

```
var i;
var n = inp.length;
for (i = 0; i < n; i++) {
    // Accessing the string alternatively from beginning and end will
    // have a major performance impact.
    print(inp.charCodeAt(i));
    print(inp.charCodeAt(n - 1 - i));
}
```

As mentioned above, these performance issues are avoided entirely for ASCII strings which behave as one would expect. More generally, Duktape provides fast paths for ASCII characters and pure ASCII strings in internal algorithms whenever applicable. This applies to algorithms such as case conversion, regexp matching, etc.

Buffer accesses

There is a fast path for reading and writing numeric indices of plain buffer values, e.g. `x = buf[123]` or `buf[123] = x`. The fast path avoids coercing the index to a string (here `"123"`) before attempting a lookup.

This fast path is not active when the base value is a Buffer object.

Object/array storage

Object properties are stored in a linear key/value list which provides stable ordering (insertion order). When an object has enough properties (internal define `DUK_HOBJECT_E_USE_HASH_LIMIT`, currently 32), a hash lookup table is also allocated to speed up property lookups. Even in this case the key ordering is retained which is a practical requirement for an Ecmascript implementation. The hash part is avoided for most objects because it increases memory footprint and doesn't significantly speed up property lookups for very small objects.

For most objects property lookup thus involves a linear comparison against the object's property table. Because properties are kept in the property table in their insertion order, properties added earlier are slightly faster to access than those added later. When the object grows large enough to gain a hash table this effect disappears.

Array elements are stored in a special "array part" to reduce memory footprint and to speed up access. Accessing an array with a numeric index officially first coerces the number to a string (e.g. `x[123]` to `x["123"]`) and then does a string key lookup; when an object has an array part no temporary string is actually created in most cases.

The array part can be "sparse", i.e. contain unmapped entries. Duktape occasionally rechecks the density of the array part, and it it becomes too sparse the array part is abandoned (current limit is roughly: if fewer than 25% of array part elements are mapped, the array part is abandoned). The array entries are then converted to ordinary object properties, with every mapped array index converted to an explicit string key (such as `"123"`), which is relatively expensive. If an array part has once been abandoned, it is never recreated even if the object would be dense enough to warrant an array part.

Elements in the array part are required to be plain properties (not accessors) and have default property attributes (writable, enumerable, and configurable). If any element deviates from this, the array part is again abandoned and array elements converted to ordinary properties.

Identifier access

Duktape has two modes for storing and accessing identifiers (function arguments, local variables, function declarations): a fast path and a slow path. The fast path is used when an identifier can be bound to a virtual machine register, i.e., a fixed index in a virtual stack frame allocated for a function. Identifier access is then simply an array lookup. The slow path is used when the fast path cannot be safely used; identifier accesses are then converted to explicit property lookups on either external or internal objects, which is more than an order of magnitude slower.

To keep identifier accesses in the fast path:

- Execute (almost all) inside Ecmascript functions, not in the top-level program or eval code: global/eval code never uses fast path identifier accesses (however, function code inside global/eval does)
- Store frequently accessed values in local variables instead of looking them up from the global object or other objects

Enumeration

When an object is enumerated, with either the `for-in` statement or `Object.keys()`, Duktape first traverses the target object and its prototype chain and forms an internal enumeration object, which contains all the enumeration keys as strings. In particular, all array indices (or character indices in case of strings) are converted and interned into string values before enumeration and they remain interned until the enumeration completes. This can be memory intensive especially if large arrays or strings are enumerated.

Note, however, that iterating a string or an array with `for-in` and expecting the array elements or string indices to be enumerated in

an ascending order is non-portable. Such behavior, while guaranteed by many implementations including Duktape, is not guaranteed by the Ecmascript standard.

Function features

Ecmascript has a lot of features which make function entry and execution quite expensive. The general goal of the Duktape Ecmascript compiler is to avoid all the troublesome features for most functions while providing full compatibility for the rest.

An ideal compiled function has all its variables and functions bound to virtual machine registers to allow fast path identifier access, avoids creation of the `arguments` object on entry, avoids creation of explicit lexical environment records upon entry and during execution, and avoids storing any lexical environment related control information such as internal identifier-to-register binding tables.

The following features have a significant impact on execution performance:

- access to the `arguments` object: requires creation of an expensive object upon function entry in case it is accessed
- a direct call to `eval()`: requires initialization of the `arguments` and full identifier binding information needs to be retained in case evaluated code needs it
- global and eval code in general: identifiers are never bound to virtual machine registers but use explicit property lookups instead

The following features have a more moderate impact:

- `try-catch-finally` statement: the dynamic binding required by the catch variable is relatively expensive
- `with` statement: the object binding required is relatively expensive
- use of bound functions, i.e. functions created with `Function.prototype.bind()`: function invocation is slowed down by handling of bound function objects and argument shuffling
- more than about 250 formal arguments, literals, and active temporaries: causes bytecode to use register shuffling

To avoid these, isolate performance critical parts into separate minimal functions which avoid using the features mentioned above.

Minimize use of temporary strings

All temporary strings are interned. It is particularly bad to accumulate strings in a loop:

```
var t = '';
for (var i = 0; i < 1024; i++) {
    t += 'x';
}
```

This will intern 1025 strings. Execution time is `O(n^2)` where `n` is the loop limit. It is better to use a temporary array instead:

```
var t = [];
for (var i = 0; i < 1024; i++) {
    t[i] = 'x';
}
t = t.join('');
```

Here, `x` will be interned once into a function constant, and each array entry simply refers to the same string, typically costing only 8 bytes per array entry. The final `Array.prototype.join()` avoids unnecessary interning and creates the final string in one go.

Avoid large non-ASCII strings if possible

Avoid operations which require access to a random character offset inside a large string containing one or more non-ASCII

characters. Such accesses require use of the internal "string cache" and may, in the worst case, require a brute force scanning of the string to find the correct byte offset corresponding to the character offset.

Case conversion and other Unicode related operations have fast paths for ASCII codepoints but fall back to a slow path for non-ASCII codepoints. The slow path is size optimized, not speed optimized, and often involve linear range matching.

Iterate over plain buffer values, not Buffer objects

Plain buffer values have a fast path when buffer contents are accessed with numeric indices. When dealing with a value which is potentially a Buffer object (not a plain buffer), get the plain buffer before iteration:

```
var b, i, n;

// Buffer object, typeof is 'object'
var bufferValue = new Duktape.Buffer('foo');
print(typeof bufferValue);  // 'object'

// Get plain buffer, if already plain, no harm
b = bufferValue.valueOf();
print(typeof b);  // always 'buffer'

n = b.length;
for (i = 0; i < n; i++) {
    print(i, b[i]);  // fast path buffer access
}
```

When creating buffers, note that `new Duktape.Buffer(x)` always creates a Buffer object, while `Duktape.Buffer(x)` returns a plain buffer value. This mimics how Ecmascript `new String()` and `String()` work. Plain buffers should be preferred whenever possible.

Avoid sparse arrays when possible

If an array becomes too sparse at any point, Duktape will abandon the array part permanently and convert array properties to explicit string keyed properties. This may happen for instance if an array is initialized with a descending index:

```
var arr = [];
for (var i = 1000; i >= 0; i--) {
    // bad: first write will abandon array part permanently
    arr[i] = i * i;
}
```

Right after the first array write the array part would contain 1001 entries with only one mapped array element. The density of the array would thus be less than 0.1%. This is way below the density limit for abandoning the array part, so the array part is abandoned immediately. At the end the array part would be 100% dense but will never be restored. Using an ascending index fixes the issue:

```
var arr = [];
for (var i = 0; i <= 1000; i++) {
    arr[i] = i * i;
}
```

Setting the `length` property of an array manually does not, by itself, cause an array part to be abandoned. To simplify a bit, the array density check compares the number of mapped elements relative to the highest used element (actually allocated size). The `length` property does not affect the check. Although setting an array length beforehand may effectively pre-allocate an array in some implementations, it has no such effect in Duktape, at least at the moment. For example:

```
var arr = [];
arr.length = 1001;  // array part not abandoned, but no speedup in Duktape
for (var i = 0; i <= 1000; i++) {
    arr[i] = i * i;
}
```

Iterate arrays with explicit indices, not a "for-in"

Because the internal enumeration object contains all (used) array indices converted to string values, avoid `for-in` enumeration of at least large arrays. As a concrete example, consider:

```
var a = [];
for (var i = 0; i < 1000000; i++) {
  a[i] = i;
}
for (var i in a) {
  // Before this loop is first entered, a million strings ("0", "1",
  // ..., "999999") will be interned.
  print(i, a[i]);
}
// The million strings become garbage collectable only here.
```

The internal enumeration object created in this example would contain a million interned string keys for "0", "1", ..., "999999". All of these keys would remain reachable for the entire duration of the enumeration. The following code would perform much better (and would be more portable, as it makes no assumptions on enumeration order):

```
var a = [];
for (var i = 0; i < 1000000; i++) {
  a[i] = i;
}
var n = a.length;
for (var i = 0; i < n; i++) {
  print(i, a[i]);
}
```

Minimize top-level global/eval code

Identifier accesses in global and eval code always use slow path instructions to ensure correctness. This is at least a few orders of magnitude slower than the fast path where identifiers are mapped to registers of a function activation.

So, this is slow:

```
for (var i = 0; i < 100; i++) {
    print(i);
}
```

Each read and write of `i` will be an explicit environment record lookup, essentially a property lookup from an internal environment record object, with the string key `i`.

Optimize by putting most code into a function:

```
function main() {
    for (var i = 0; i < 100; i++) {
        print(i);
    }
}
main();
```

Here, `i` will be mapped to a function register, and each access will be a simple register reference (basically a pointer to a tagged value), which is much faster than the slow path.

If you don't want to name an explicit function, use:

```
(function() {
    var i;

    for (i = 0; i < 100; i++) {
      print(i);
    }
})();
```

Eval code provides an implicit return value which also has a performance impact. Consider, for instance, the following:

```
var res = eval("if (4 > 3) { 'foo'; } else { 'bar'; }");
print(res);  // prints 'foo'
```

To support such code the compiler emits bytecode to store a statement's implicit return value to a temporary register in case it is needed. These instructions slow down execution and increase bytecode size unnecessarily.

Prefer local variables over external ones

When variables are bound to virtual machine registers, identifier lookups are much faster than using explicit property lookups on the global object or on other objects.

When an external value or function is required multiple times, copy it to a local variable instead:

```
function slow(x) {
    var i;

    // 'x.length' is an explicit property lookup and happens on every loop
    for (i = 0; i < x.length; i++) {
        // 'print' causes a property lookup to the global object
        print(x[i]);
    }
}

function fast(x) {
    var i;
    var n = x.length;
    var p = print;

    // every access in the loop now happens through register-bound identifiers
    for (i = 0; i < n; i++) {
        p(x[i]);
    }
}
```

Use such optimizations only where it matters, because they often reduce code readability.

Portability

Platforms and compilers

The table below summarizes the platforms and compilers which Duktape is known to work on, with portability notes where appropriate. This is **not an exhaustive list** of supported/unsupported platforms, rather a list of what is known to work (and not to work). Platform and compiler specific issues are discussed in more detail below the table.

Operating system	Compiler	Processor	Notes
Linux	GCC	x86	No known issues.
Linux	GCC	x64	No known issues.
Linux	GCC	ARM	No known issues.
Linux	GCC	MIPS	No known issues.
Linux	Clang	x86	No known issues.
Linux	Clang	x64	No known issues.
Linux	Clang	ARM	No known issues.
Linux	Clang	MIPS	No known issues.
Linux	TCC	x64	Zero sign issues (see below).
FreeBSD	Clang	x86	Aliasing issues with clang 3.3 on 64-bit FreeBSD, `-m32`, and packed `duk_tval` (see below).
FreeBSD	Clang	x64	No known issues.
NetBSD	GCC	x86	No known issues (NetBSD 6.0). There are some `pow()` function

			incompatibilities on NetBSD, but there is a workaround for them.
OpenBSD	GCC	x86	No known issues (FreeBSD 5.4).
Windows	MinGW	x86	`-std=c99` recommended, only ISO 8601 date format supported (no platform specific format).
Windows	MinGW-w64	x64	`-m64`, `-std=c99` recommended, only ISO 8601 date format supported (no platform specific format).
Windows	MSVC (Visual Studio Express 2010)	x86	Only ISO 8601 date format supported (no platform specific format).
Windows	MSVC (Visual Studio Express 2013 for Windows Desktop)	x64	Only ISO 8601 date format supported (no platform specific format).
Android	GCC (Android NDK)	ARM	No known issues.
OSX	Clang	x64	Tested on OSX 10.9.2 with XCode.
Darwin	GCC	x86	No known issues.
QNX	GCC	x86	`-std=c99` required. Architectures other than x86 should also work.
AmigaOS	VBCC	M68K	Requires some preprocessor defines, datetime resolution limited to full seconds.
TOS (Atari ST)	VBCC	M68K	Requires some preprocessor defines, datetime resolution limited to full seconds.
Emscripten	Emscripten	n/a	Requires additional options, see below. At least V8/NodeJs works.
Adobe Flash Runtime	CrossBridge (GCC-4.2 with Flash backend)	n/a	`-std=c99` recommended, may need `-jvmopt=-Xmx1G` if running 32-bit Java. Tested with CrossBridge 1.0.1 on 64-bit Windows 7.

Clang

Clang 3.3 on FreeBSD has some aliasing issues (at least) when using `-m32` and when Duktape ends up using a packed `duk_tval` value representation type. You can work around the problem by defining `DUK_OPT_NO_PACKED_TVAL` to disable packed value type. The problem does not appear in all clang versions. Duktape self tests cover this issue (define `DUK_OPT_SELF_TESTS` when compiling). See internal test file `misc/clang_aliasing.c`.

TCC

TCC has zero sign handling issues; Duktape mostly works but zero sign is not handled correctly. This results in Ecmascript non-compliance, for instance `1/-0` evaluates to `Infinity`, not `-Infinity` as it should.

VBCC (AmigaOS / TOS)

VBCC doesn't appear to provide OS or processor defines. To compile for M68K AmigaOS or TOS you must:

- Define `__MC68K__` manually.
- Define either `AMIGA` or `__TOS__` manually.

Datetime resolution is limited to full seconds only when using VBCC on AmigaOS or TOS.

Emscripten

Needs a set of `emcc` options. When executed with V8, the following seem to work:

- `-DEMSCRIPTEN`: **mandatory option**, needed by Duktape to detect Emscripten. Without this Duktape may use unaligned accesses which Emscripten does not allow. This results in odd and inconsistent behavior, and is not necessarily caught by Duktape self tests.
- `-std=c99`
- `-O2`
- `-s ASM_JS=0`
- `-s MAX_SETJMPS=1000`
- `-s OUTLINING_LIMIT=20000`

Dukweb is compiled using Emscripten, so you can also check out the Duktape git repository to see how Dukweb is compiled.

Using Duktape from a C++ program

To use Duktape from a C++ program, simply compile Duktape in plain C and use `duktape.h` normally in your C++ program; `duktape.h` contains the necessary glue to make this work. Specifically, it contains `extern "C" { ... }` to avoid name mangling issues.

Currently Duktape itself cannot be compiled in C++ mode. This is under work but is not a trivial issue because many of the compiler defines and headers are different (especially for pre C99/C++11).

Limitations

- The `int` type is assumed to be at least 32 bits. This is incorrect even on some platforms which provide a 32-bit type.
- Pointer less-than/greater-than comparisons are expected to work like pointers were unsigned. This is incorrect on some platforms.
- On platforms requiring aligned accesses, Duktape guarantees 4-byte alignment. In particular, 64-bit integers and IEEE double values are not guaranteed to be 8-byte aligned. This is not always correct.

Troubleshooting

- Compile in C mode if possible. Although C++ compilation now works, it isn't as portable as C compilation.
- Enable C99 mode if possible (`-std=c99` or similar). Type detection without C99 is less reliable than with C99.
- If Duktape compiles but doesn't seem to work correctly, enable self tests with `DUK_OPT_SELF_TESTS`. Self tests detect some compiler and platform issues which cannot be caught compile time.
- If the target platform has specific alignment requirements and Duktape doesn't autodetect the platform correctly, you may need to provide either `DUK_OPT_FORCE_ALIGN=4` or `DUK_OPT_FORCE_ALIGN=8`. The alignment number should match whatever alignment is needed for IEEE doubles and 64-bit integer values.
- If compilation fails in endianness detection, Duktape probably doesn't (yet) support the platform specific endianness headers of your platform. Such headers are unfortunately non-standardized, so endianness detection is a common (and usually trivial) portability issue on custom platforms. Use `DUK_OPT_FORCE_BYTEORDER` to force endianness as a workaround. If you know how the endianness detection should work on your platform, please send an e-mail about the issue or contribute a patch.
- Another typical portability issue on new platforms is the Date built-in, which requires a few platform specific functions for dealing with date and time. Often existing Date functions are sufficient but platform detection in `duk_features.h` does not yet handle the target platform correctly. This is usually trivial to fix; please contribute a patch if you do so. At other times the platform has no standard time APIs (like POSIX). In this case you'll need to add a few platform specific Date functions into `duk_bi_date.c`, and implement platform detection into `duk_features.h`; again, please contribute a patch if you do so. You can look at `duk_bi_date.c` for POSIX and Windows Date API examples.

Compatibility

This section discussed Duktape compatibility with Ecmascript dialects, extensions, frameworks, and test suites.

Ecmascript E5 / E5.1

The main compatibility goal of Duktape is to be Ecmascript E5/E5.1 compatible. Current level of compatibility should be quite high.

Ecmascript E3

There is no effort to maintain Ecmascript E3 compatibility, other than required by the E5/E5.1 specification.

CoffeeScript

CoffeeScript compiles to JavaScript which should be compatible with Duktape. There are no known compatibility issues.

Some CoffeeScript examples are included in the distributable. Simply run `make` in `examples/coffee/`. For instance, `hello.coffee`:

```
print 'Hello world!'
print 'version: ' + Duktape.version
```

compiles to:

```
(function() {

  print('Hello world!');

  print('version: ' + Duktape.version);

}).call(this);
```

Coco

Like CoffeeScript, Coco compiles to Javascript. There are no known issues.

LiveScript

Like CoffeeScript, LiveScript compiles to Javascript. There are no known issues.

Underscore.js

Underscore.js provides a lot of useful utilities to plain Ecmascript. Duktape passes almost all of Underscore's test cases, see underscore-status.txt for current compatibility status.

Test262

test262 is a test suite for testing E5.1 compatibility, although it includes also tests outside of standard E5.1. Duktape passes almost all of test262 cases, see test262-status.txt for current compatibility status.

Asm.js

asm.js is a "strict subset of JavaScript that can be used as a low-level, efficient target language for compilers". As a subset of JavaScript, functions using asm.js type annotations should be fully compatible with Duktape. However, Duktape has no specific

support for asm.js and won't optimize asm.js code. In fact, asm.js code will generate unnecessary bytecode and execute slower than normal Ecmascript code. The `"use asm"` directive specified by asm.js is ignored by Duktape. Also, because there is not typed array support yet, no "heap object" can be provided.

Emscripten

Emscripten compiles C/C++ into Javascript. Duktape is currently Emscripten compatible except for:

- Duktape doesn't yet have typed arrays, so give `emcc` the option `-s USE_TYPED_ARRAYS=0` to disable their use.

Performance is somewhat limited as Duktape is an interpreted engine. Lack of typed array support also forces Emscripten to use a much slower model for emulating application memory. Large programs may fail due to Duktape compiler running out of virtual registers. See emscripten-status.txt for current compatibility status.

Duktape itself compiles with Emscripten, and it is possible to run Duktape inside a web page for instance, see Dukweb REPL.

Lua.js

lua.js translates Lua code to Javascript. There are no known issues in running the generated Javascript, except that Duktape doesn't provide `console.log` which lua.js expects. This is easy to remedy, e.g. by prepending the following:

```
console = { log: function() { print(Array.prototype.join.call(arguments, ' ')); } };
```

JS-Interpreter

JS-Interpreter interprets Javascript in Javascript. JS-Interpreter works with Duktape, except that Duktape doesn't provide `window` which JS-Interpreter expects. This can be fixed by prepending:

```
window = {};
```

Limitations

The following is a list of known limitations of the current implementation. Limitations include shortcomings from a semantics perspective, performance limitations, and implementation limits (which are inevitable).

Trivial bugs are not listed unless they are "long term bugs".

No re-entrancy

A single Duktape heap, i.e. contexts sharing the same garbage collector, is **not re-entrant**. Only one C/C++ thread can call Duktape APIs at a time for a particular Duktape heap (although the calling thread can change over time).

String and buffer limits

The internal representation allows a maximum length of 2**31-1 (0x7fffffff) bytes (not characters) for strings. 16-bit codepoints encode into 3 bytes of UTF-8 in the worst case, so the maximum string length which is guaranteed to work is about 0.7G characters.

Buffer values are also limited to 2**31-1 (0x7fffffff) bytes.

Property limits

An object can have at most `DUK_HOBJECT_MAX_PROPERTIES` (an internal define). Currently this limit is 0x7fffffff.

Array limits

When array item indices go over the 2**31-1 limit (0x7fffffff), Duktape has some known bugs with array semantics.

Regexp quantifier over empty match

The regexp engine gets stuck when a quantifier is used over an empty match but eventually bails out with an internal recursion (or execution step) limit. For instance, the following should produce a "no match" result but hits an internal recursion limit instead:

```
$ duk
duk> t = /(x*)*/.exec('y');
RangeError: regexp executor recursion limit
        duk_regexp_executor.c:145
        exec (null) native strict preventsyield
        global input:1 preventsyield
```

Regexp dollar escape

The Duktape RegExp syntax allows dollar escaping (e.g. `/\$/`) even though it is not allowed by the E5.1 specification. RegExp dollar escapes are used in existing Ecmascript code quite widely.

Invalid stack indices

The internal implementation for some stack operations (like `duk_set_top()` uses pointer arithmetic. On 32-bit platforms the pointer arithmetic may wrap and work in unexpected ways if stack index values are large enough (e.g. 0x20000000 on a 32-bit platform with 8-byte packed value type).

Unicode case conversion is not locale or context sensitive

E5 Sections 15.5.4.16 to 15.5.4.19 require context and locale processing of Unicode SpecialCasing.txt. However, Duktape doesn't currently have a notion of "current locale".

Array performance when using non-default property attributes

All array elements are expected to be writable, enumerable, and configurable (default property attributes for new properties). If this assumption is violated, even temporarily, the entire "array part" of an object is abandoned permanently and array entries are moved to the "entry part". This involves interning all used array indices as explicit string keys (e.g. "0", "1", etc). This is not a compliance concern, but degrades performance.

Global/eval code is slower than function code

Bytecode generated for global and eval code cannot assign variables statically to registers, and will use explicit name-based variable read/write accesses. Bytecode generated for function code doesn't have this limitation; most variables are assigned statically to registers and direct register references are used used to access them.

This is a minor issue unless you spend a lot of time running top-level global/eval code. The workaround is simple: put code in a function which you call from the top level; for instance:

```
function main() {
    // ...
}
main();
```

There is also a common idiom of using an anonymous function for this purpose:

```
(function () {
    // ...
})();
```

Function temporaries may be live for garbage collection longer than expected

Ecmascript functions are compiled into bytecode with a fixed set of registers. Some registers are reserved for arguments and variable bindings while others are used as temporaries. All registers are considered live from a garbage collection perspective, even temporary registers containing old values which the function actually cannot reference any more. Such temporaries are considered reachable until they are overwritten by the evaluation of another expression or until the function exits. Function exit is the only easily predicted condition to ensure garbage collection.

If you have a function which remains running for a very long time, it should contain the bare minimum of variables and temporaries that could remain live. For instance, you can structure code like:

```
function runOnce() {
    // run one iteration, lots of temporaries
}

function foreverLoop() {
    for (;;) {
        runOnce();
    }
}
```

This is typically not an issue if there are no long-running functions.

Function instances are garbage collected only by mark-and-sweep

Every Ecmascript function instance is, by default, in a reference loop with an automatic prototype object created for the function. The function instance's `prototype` property points to the prototype object, and the prototype's `constructor` property points back to the function instance. Only mark-and-sweep is able to collect these reference loops at the moment. If you build with reference counting only, function instances may appear to leak memory; the memory will be released when the relevant heap is destroyed.

You can break the reference loops manually (although this is a bit cumbersome):

```
var f = function() { };
var g = function() { };
var h = function() { };
Duktape.fin(f, function() { print('finalizer for f'); });
Duktape.fin(g, function() { print('finalizer for g'); });
Duktape.fin(h, function() { print('finalizer for h'); });

// not collected until heap destruction in a reference counting only build
f = null;            // not collected immediately

// break cycle by deleting 'prototype' reference (alternative 1)
g.prototype = null;
g = null;            // collected immediately, finalizer runs

// break cycle by deleting 'constructor' reference (alternative 2)
h.prototype.constructor = null;
h = null;            // collected immediately, finalizer runs

// no-op with refcount only, with mark-and-sweep finalizer for 'f' runs
Duktape.gc();
```

For internal technical reasons, named function expressions are also in a reference loop with an internal environment record object. This loop cannot be broken from user code and only mark-and-sweep can collect such functions. Ordinary function declarations and anonymous functions don't have this limitation. Example:

```
var fn = function myfunc() {
    // myfunc is in reference loop with an internal environment record,
    // and can only be collected with mark-and-sweep.
}
```

These issues can be avoided by compiling Duktape with mark-and-sweep enabled (which is the default).

Non-standard function 'caller' property limitations

When `DUK_OPT_FUNC_NONSTD_CALLER_PROPERTY` is given, Duktape updates the `caller` property of non-strict function instances similarly to e.g. V8 and Spidermonkey. There are a few limitations, though:

- When a (non-strict) function is called from eval code, Duktape sets `caller` to `null` if the eval code is non-strict, and `eval` (reference to the eval built-in function) if the eval code is strict. This deviates from e.g. V8 behavior.
- Coroutines and `caller` don't mix well: `caller` may be left in a non-`null` state even after coroutine call stacks have been fully unwound. Also, if a coroutine is garbage collected before its call stack is unwound, the `caller` property of functions in its call stack will not get updated now.

See the internal `test-bi-function-nonstd-caller-prop.js` test case for further details.

Comparison to Lua

Duktape borrows a lot from Lua conceptually. Below are a few notes on what's different in Duktape compared to Lua. This may be

useful if you're already familiar with Lua.

Array and stack indices are zero-based

All array and stack indices are zero-based, not one-based as in Lua. So, bottom of stack is 0, second element from bottom is 1, and top element is -1. Because 0 is no longer available to denote an invalid/non-existent element, the constant `DUK_INVALID_INDEX` is used instead in Duktape.

String indices are also zero-based, and slices are indicated with an inclusive start index and an exclusive end index (i.e. [start,end[). In Lua, slices are indicated with inclusive indices (i.e. [start,end]).

Object type represents functions and threads

In Lua functions and threads are a separate type from objects. In Duktape the object type is used for plain objects, Ecmascript and native functions, and threads (coroutines). As a result, all of these have a mutable and extensible set of properties.

Lua userdata and lightuserdata

The concept closest to Lua `userdata` is the Duktape `buffer` type, with the following differences:

- Duktape buffers can be resizable, Lua userdata values cannot. If a Duktape buffer is resizable, its data pointer is no longer guaranteed to be stable.
- Duktape buffers are raw byte arrays without any properties, Lua userdata objects can store an environment reference.

Lua `lightuserdata` and Duktape `pointer` are essentially the same.

If you need to associate properties with a Duktape buffer, you can use an actual object and have the buffer as its property. You can then add a finalizer to the object to free any resources related to the buffer. This works reasonably well as long as nothing else holds a reference to the buffer. If this were the case, the buffer could get used after the object had already been finalized. To safeguard against this, the native C structure should have a flag indicating whether the data structure is open or closed. This is good practice anyway for robust native code.

Garbage collection

Duktape has a combined reference counting and non-incremental mark-and-sweep garbage collector (mark-and-sweep is needed only for reference cycles). Collection pauses can be avoided by disabling voluntary mark-and-sweep passes (`DUK_OPT_NO_VOLUNTARY_GC`). Lua has an incremental collector with no pauses, but has no reference counting.

Duktape has an emergency garbage collector. Lua 5.2 has an emergency garbage collector while Lua 5.1 does not (there is an emergency GC patch though).

Finalizers

Duktape has finalizer supported. Lua 5.2 has finalizer support while Lua 5.1 does not.

duk_safe_call() vs. lua_cpcall()

`duk_safe_call()` is a protected C function call which operates in the existing value stack frame. The function call is not visible on the call stack all.

`lua_cpcall()` creates a new stack frame.

Bytecode use

Duktape Ecmascript function bytecode is currently a purely internal matter. Code cannot currently be loaded from an external pre-compiled bytecode file. Similarly, there is no equivalent to e.g. `lua_dump()`.

Metatables

There is currently no equivalent of Lua metatables in Duktape (or Ecmascript). The ES6 proxy object concept will most likely provide similar functionality at some point.

lua_next() vs. duk_next()

`lua_next()` replaces the previous key and value with a new pair, while `duk_next()` does not; the caller needs to explicitly pop the key and/or value.

Raw accessors

There is no equivalent to Lua raw table access functions like `lua_rawget`. One can use the following Ecmascript built-ins for a similar effect (though not with respect to performance): Object.getOwnPropertyDescriptor (O, P), Object.defineProperty (O, P, Attributes).

Coroutines

There are no primitives for coroutine control in the Duktape API (Lua API has e.g. `lua_resume`). Coroutines can only be controlled using the functions exposed by the `Duktape` built-in. Further, Duktape has quite many coroutine yield restrictions now; for instance, coroutines cannot yield from inside constructor calls or getter/setter calls.

Multiple return values

Lua supports multiple return values, Duktape (or Ecmascript) currently doesn't. This may change with Ecmascript E6, which has a syntax for multiple value returns. The Duktape/C API reserves return values above 1 so that they may be later used for multiple return values.

Unicode

Lua has no built-in Unicode support (strings are byte strings), while Duktape has support for 16-bit Unicode as part of Ecmascript compliance.

Streaming compilation

Lua has a streaming compilation API which is good when code is read from the disk or perhaps decompressed on-the-fly. Duktape currently does not support streaming compilation because it needs multiple passes over the source code.