Node-API

> *Stability: 2 - Stable*

Node-API (formerly N-API) is an API for building native Addons. It is independent from the underlying JavaScript runtime (for example, V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate addons from changes in the underlying JavaScript engine and allow modules compiled for one major version to run on later major versions of Node.js without recompilation. The ABI Stability guide provides a more in-depth explanation.

Addons are built/packaged with the same approach/tools outlined in the section titled C++ Addons. The only difference is the set of APIs that are used by the native code. Instead of using the V8 or Native Abstractions for Node.js APIs, the functions available in Node-API are used.

APIs exposed by Node-API are generally used to create and manipulate JavaScript values. Concepts and operations generally map to ideas specified in the ECMA-262 Language Specification. The APIs have the following properties:

- All Node-API calls return a status code of type `napi_status`. This status indicates whether the API call succeeded or failed.
- The API's return value is passed via an out parameter.
- All JavaScript values are abstracted behind an opaque type named `napi_value`.
- In case of an error status code, additional information can be obtained using `napi_get_last_error_info`. More information can be found in the error handling section Error handling.

Node-API is a C API that ensures ABI stability across Node.js versions and different compiler levels. A C++ API can be easier to use. To support using C++, the project maintains a C++ wrapper module called `node-addon-api`. This wrapper provides an inlineable C++ API. Binaries built with `node-addon-api` will depend on the symbols for the Node-API C-based functions exported by Node.js. `node-addon-api` is a more efficient way to write code that calls Node-API. Take, for example, the following `node-addon-api` code. The first section shows the `node-addon-api` code and the second section shows what actually gets used in the addon.

```
Object obj = Object::New(env);
obj["foo"] = String::New(env, "bar");
```

```
napi_status status;
napi_value object, string;
status = napi_create_object(env, &object);
if (status != napi_ok) {
  napi_throw_error(env, ...);
  return;
}

status = napi_create_string_utf8(env, "bar", NAPI_AUTO_LENGTH, &string);
if (status != napi_ok) {
  napi_throw_error(env, ...);
  return;
}

status = napi_set_named_property(env, object, "foo", string);
```

```
if (status != napi_ok) {
  napi_throw_error(env, ...);
  return;
}
```

The end result is that the addon only uses the exported C APIs. As a result, it still gets the benefits of the ABI stability provided by the C API.

When using `node-addon-api` instead of the C APIs, start with the API docs for `node-addon-api`.

The Node-API Resource offers an excellent orientation and tips for developers just getting started with Node-API and `node-addon-api`.

Implications of ABI stability

Although Node-API provides an ABI stability guarantee, other parts of Node.js do not, and any external libraries used from the addon may not. In particular, none of the following APIs provide an ABI stability guarantee across major versions:

- the Node.js C++ APIs available via any of

```
#include <node.h>
#include <node_buffer.h>
#include <node_version.h>
#include <node_object_wrap.h>
```

- the libuv APIs which are also included with Node.js and available via

```
#include <uv.h>
```

- the V8 API available via

```
#include <v8.h>
```

Thus, for an addon to remain ABI-compatible across Node.js major versions, it must use Node-API exclusively by restricting itself to using

```
#include <node_api.h>
```

and by checking, for all external libraries that it uses, that the external library makes ABI stability guarantees similar to Node-API.

Building

Unlike modules written in JavaScript, developing and deploying Node.js native addons using Node-API requires an additional set of tools. Besides the basic tools required to develop for Node.js, the native addon developer requires a toolchain that can compile C and C++ code into a binary. In addition, depending upon how the native addon is deployed, the *user* of the native addon will also need to have a C/C++ toolchain installed.

For Linux developers, the necessary C/C++ toolchain packages are readily available. GCC is widely used in the Node.js community to build and test across a variety of platforms. For many developers, the LLVM compiler infrastructure is also a good choice.

For Mac developers, Xcode offers all the required compiler tools. However, it is not necessary to install the entire Xcode IDE. The following command installs the necessary toolchain:

```
xcode-select --install
```

For Windows developers, Visual Studio offers all the required compiler tools. However, it is not necessary to install the entire Visual Studio IDE. The following command installs the necessary toolchain:

```
npm install --global windows-build-tools
```

The sections below describe the additional tools available for developing and deploying Node.js native addons.

Build tools

Both the tools listed here require that *users* of the native addon have a C/C++ toolchain installed in order to successfully install the native addon.

node-gyp

node-gyp is a build system based on the gyp-next fork of Google's GYP tool and comes bundled with npm. GYP, and therefore node-gyp, requires that Python be installed.

Historically, node-gyp has been the tool of choice for building native addons. It has widespread adoption and documentation. However, some developers have run into limitations in node-gyp.

CMake.js

CMake.js is an alternative build system based on CMake.

CMake.js is a good choice for projects that already use CMake or for developers affected by limitations in node-gyp.

Uploading precompiled binaries

The three tools listed here permit native addon developers and maintainers to create and upload binaries to public or private servers. These tools are typically integrated with CI/CD build systems like Travis CI and AppVeyor to build and upload binaries for a variety of platforms and architectures. These binaries are then available for download by users who do not need to have a C/C++ toolchain installed.

node-pre-gyp

node-pre-gyp is a tool based on node-gyp that adds the ability to upload binaries to a server of the developer's choice. node-pre-gyp has particularly good support for uploading binaries to Amazon S3.

prebuild

prebuild is a tool that supports builds using either node-gyp or CMake.js. Unlike node-pre-gyp which supports a variety of servers, prebuild uploads binaries only to GitHub releases. prebuild is a good choice for GitHub projects using CMake.js.

prebuildify

prebuildify is a tool based on node-gyp. The advantage of prebuildify is that the built binaries are bundled with the native module when it's uploaded to npm. The binaries are downloaded from npm and are immediately available to the module user when the native module is installed.

Usage

In order to use the Node-API functions, include the file `node_api.h` which is located in the src directory in the node development tree:

```
#include <node_api.h>
```

This will opt into the default `NAPI_VERSION` for the given release of Node.js. In order to ensure compatibility with specific versions of Node-API, the version can be specified explicitly when including the header:

```
#define NAPI_VERSION 3
#include <node_api.h>
```

This restricts the Node-API surface to just the functionality that was available in the specified (and earlier) versions.

Some of the Node-API surface is experimental and requires explicit opt-in:

```
#define NAPI_EXPERIMENTAL
#include <node_api.h>
```

In this case the entire API surface, including any experimental APIs, will be available to the module code.

Node-API version matrix

Node-API versions are additive and versioned independently from Node.js. Version 4 is an extension to version 3 in that it has all of the APIs from version 3 with some additions. This means that it is not necessary to recompile for new versions of Node.js which are listed as supporting a later version.

	1	2	3
v6.x			v6.14.2*
v8.x	v8.6.0**	v8.10.0*	v8.11.2
v9.x	v9.0.0*	v9.3.0*	v9.11.0*
≥ v10.x	all releases	all releases	all releases

	4	5	6	7	8
v10.x	v10.16.0	v10.17.0	v10.20.0	v10.23.0	
v11.x	v11.8.0				
v12.x	v12.0.0	v12.11.0	v12.17.0	v12.19.0	v12.22.0
v13.x	v13.0.0	v13.0.0			

v14.x	v14.0.0	v14.0.0	v14.0.0	v14.12.0	v14.17.0
v15.x	v15.0.0	v15.0.0	v15.0.0	v15.0.0	v15.12.0
v16.x	v16.0.0	v16.0.0	v16.0.0	v16.0.0	v16.0.0

* Node-API was experimental.

** Node.js 8.0.0 included Node-API as experimental. It was released as Node-API version 1 but continued to evolve until Node.js 8.6.0. The API is different in versions prior to Node.js 8.6.0. We recommend Node-API version 3 or later.

Each API documented for Node-API will have a header named `added in:`, and APIs which are stable will have the additional header `Node-API version:`. APIs are directly usable when using a Node.js version which supports the Node-API version shown in `Node-API version:` or higher. When using a Node.js version that does not support the `Node-API version:` listed or if there is no `Node-API version:` listed, then the API will only be available if `#define NAPI_EXPERIMENTAL` precedes the inclusion of `node_api.h` or `js_native_api.h`. If an API appears not to be available on a version of Node.js which is later than the one shown in `added in:` then this is most likely the reason for the apparent absence.

The Node-APIs associated strictly with accessing ECMAScript features from native code can be found separately in `js_native_api.h` and `js_native_api_types.h`. The APIs defined in these headers are included in `node_api.h` and `node_api_types.h`. The headers are structured in this way in order to allow implementations of Node-API outside of Node.js. For those implementations the Node.js specific APIs may not be applicable.

The Node.js-specific parts of an addon can be separated from the code that exposes the actual functionality to the JavaScript environment so that the latter may be used with multiple implementations of Node-API. In the example below, `addon.c` and `addon.h` refer only to `js_native_api.h`. This ensures that `addon.c` can be reused to compile against either the Node.js implementation of Node-API or any implementation of Node-API outside of Node.js.

`addon_node.c` is a separate file that contains the Node.js specific entry point to the addon and which instantiates the addon by calling into `addon.c` when the addon is loaded into a Node.js environment.

```
// addon.h
#ifndef _ADDON_H_
#define _ADDON_H_
#include <js_native_api.h>
napi_value create_addon(napi_env env);
#endif  // _ADDON_H_
```

```
// addon.c
#include "addon.h"

#define NAPI_CALL(env, call)                                      \
  do {                                                            \
    napi_status status = (call);                                  \
    if (status != napi_ok) {                                      \
      const napi_extended_error_info* error_info = NULL;          \
      napi_get_last_error_info((env), &error_info);               \
      const char* err_message = error_info->error_message;        \
```

```
      bool is_pending;                                                \
      napi_is_exception_pending((env), &is_pending);                  \
      if (!is_pending) {                                              \
        const char* message = (err_message == NULL)                   \
            ? "empty error message"                                   \
            : err_message;                                            \
        napi_throw_error((env), NULL, message);                       \
        return NULL;                                                  \
      }                                                               \
    }                                                                 \
  } while(0)

static napi_value
DoSomethingUseful(napi_env env, napi_callback_info info) {
  // Do something useful.
  return NULL;
}

napi_value create_addon(napi_env env) {
  napi_value result;
  NAPI_CALL(env, napi_create_object(env, &result));

  napi_value exported_function;
  NAPI_CALL(env, napi_create_function(env,
                                      "doSomethingUseful",
                                      NAPI_AUTO_LENGTH,
                                      DoSomethingUseful,
                                      NULL,
                                      &exported_function));

  NAPI_CALL(env, napi_set_named_property(env,
                                         result,
                                         "doSomethingUseful",
                                         exported_function));

  return result;
}
```

```
// addon_node.c
#include <node_api.h>
#include "addon.h"

NAPI_MODULE_INIT() {
  // This function body is expected to return a `napi_value`.
  // The variables `napi_env env` and `napi_value exports` may be used within
  // the body, as they are provided by the definition of `NAPI_MODULE_INIT()`.
  return create_addon(env);
}
```

Environment life cycle APIs

Section 8.7 of the ECMAScript Language Specification defines the concept of an "Agent" as a self-contained environment in which JavaScript code runs. Multiple such Agents may be started and terminated either concurrently or in sequence by the process.

A Node.js environment corresponds to an ECMAScript Agent. In the main process, an environment is created at startup, and additional environments can be created on separate threads to serve as worker threads. When Node.js is embedded in another application, the main thread of the application may also construct and destroy a Node.js environment multiple times during the life cycle of the application process such that each Node.js environment created by the application may, in turn, during its life cycle create and destroy additional environments as worker threads.

From the perspective of a native addon this means that the bindings it provides may be called multiple times, from multiple contexts, and even concurrently from multiple threads.

Native addons may need to allocate global state which they use during their entire life cycle such that the state must be unique to each instance of the addon.

To this end, Node-API provides a way to allocate data such that its life cycle is tied to the life cycle of the Agent.

`napi_set_instance_data`

```
napi_status napi_set_instance_data(napi_env env,
                                   void* data,
                                   napi_finalize finalize_cb,
                                   void* finalize_hint);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] data` : The data item to make available to bindings of this instance.
- `[in] finalize_cb` : The function to call when the environment is being torn down. The function receives `data` so that it might free it. `napi_finalize` provides more details.
- `[in] finalize_hint` : Optional hint to pass to the finalize callback during collection.

Returns `napi_ok` if the API succeeded.

This API associates `data` with the currently running Agent. `data` can later be retrieved using `napi_get_instance_data()` . Any existing data associated with the currently running Agent which was set by means of a previous call to `napi_set_instance_data()` will be overwritten. If a `finalize_cb` was provided by the previous call, it will not be called.

`napi_get_instance_data`

```
napi_status napi_get_instance_data(napi_env env,
                                   void** data);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[out] data` : The data item that was previously associated with the currently running Agent by a call to `napi_set_instance_data()` .

Returns `napi_ok` if the API succeeded.

This API retrieves data that was previously associated with the currently running Agent via `napi_set_instance_data()`. If no data is set, the call will succeed and `data` will be set to `NULL`.

Basic Node-API data types

Node-API exposes the following fundamental datatypes as abstractions that are consumed by the various APIs. These APIs should be treated as opaque, introspectable only with other Node-API calls.

`napi_status`

Integral status code indicating the success or failure of a Node-API call. Currently, the following status codes are supported.

```
typedef enum {
  napi_ok,
  napi_invalid_arg,
  napi_object_expected,
  napi_string_expected,
  napi_name_expected,
  napi_function_expected,
  napi_number_expected,
  napi_boolean_expected,
  napi_array_expected,
  napi_generic_failure,
  napi_pending_exception,
  napi_cancelled,
  napi_escape_called_twice,
  napi_handle_scope_mismatch,
  napi_callback_scope_mismatch,
  napi_queue_full,
  napi_closing,
  napi_bigint_expected,
  napi_date_expected,
  napi_arraybuffer_expected,
  napi_detachable_arraybuffer_expected,
  napi_would_deadlock,  /* unused */
} napi_status;
```

If additional information is required upon an API returning a failed status, it can be obtained by calling `napi_get_last_error_info`.

`napi_extended_error_info`

```
typedef struct {
  const char* error_message;
  void* engine_reserved;
  uint32_t engine_error_code;
  napi_status error_code;
} napi_extended_error_info;
```

- `error_message` : UTF8-encoded string containing a VM-neutral description of the error.
- `engine_reserved` : Reserved for VM-specific error details. This is currently not implemented for any VM.
- `engine_error_code` : VM-specific error code. This is currently not implemented for any VM.
- `error_code` : The Node-API status code that originated with the last error.

See the Error handling section for additional information.

`napi_env`

`napi_env` is used to represent a context that the underlying Node-API implementation can use to persist VM-specific state. This structure is passed to native functions when they're invoked, and it must be passed back when making Node-API calls. Specifically, the same `napi_env` that was passed in when the initial native function was called must be passed to any subsequent nested Node-API calls. Caching the `napi_env` for the purpose of general reuse, and passing the `napi_env` between instances of the same addon running on different `Worker` threads is not allowed. The `napi_env` becomes invalid when an instance of a native addon is unloaded. Notification of this event is delivered through the callbacks given to `napi_add_env_cleanup_hook` and `napi_set_instance_data` .

`napi_value`

This is an opaque pointer that is used to represent a JavaScript value.

`napi_threadsafe_function`

This is an opaque pointer that represents a JavaScript function which can be called asynchronously from multiple threads via `napi_call_threadsafe_function()` .

`napi_threadsafe_function_release_mode`

A value to be given to `napi_release_threadsafe_function()` to indicate whether the thread-safe function is to be closed immediately (`napi_tsfn_abort`) or merely released (`napi_tsfn_release`) and thus available for subsequent use via `napi_acquire_threadsafe_function()` and `napi_call_threadsafe_function()` .

```
typedef enum {
  napi_tsfn_release,
  napi_tsfn_abort
} napi_threadsafe_function_release_mode;
```

`napi_threadsafe_function_call_mode`

A value to be given to `napi_call_threadsafe_function()` to indicate whether the call should block whenever the queue associated with the thread-safe function is full.

```
typedef enum {
  napi_tsfn_nonblocking,
  napi_tsfn_blocking
} napi_threadsafe_function_call_mode;
```

Node-API memory management types

`napi_handle_scope`

This is an abstraction used to control and modify the lifetime of objects created within a particular scope. In general, Node-API values are created within the context of a handle scope. When a native method is called from JavaScript, a default handle scope will exist. If the user does not explicitly create a new handle scope, Node-API values will be created in the default handle scope. For any invocations of code outside the execution of a native method (for instance, during a libuv callback invocation), the module is required to create a scope before invoking any functions that can result in the creation of JavaScript values.

Handle scopes are created using `napi_open_handle_scope` and are destroyed using `napi_close_handle_scope`. Closing the scope can indicate to the GC that all `napi_value`s created during the lifetime of the handle scope are no longer referenced from the current stack frame.

For more details, review the Object lifetime management.

`napi_escapable_handle_scope`

Escapable handle scopes are a special type of handle scope to return values created within a particular handle scope to a parent scope.

`napi_ref`

This is the abstraction to use to reference a `napi_value`. This allows for users to manage the lifetimes of JavaScript values, including defining their minimum lifetimes explicitly.

For more details, review the Object lifetime management.

`napi_type_tag`

A 128-bit value stored as two unsigned 64-bit integers. It serves as a UUID with which JavaScript objects can be "tagged" in order to ensure that they are of a certain type. This is a stronger check than `napi_instanceof`, because the latter can report a false positive if the object's prototype has been manipulated. Type-tagging is most useful in conjunction with `napi_wrap` because it ensures that the pointer retrieved from a wrapped object can be safely cast to the native type corresponding to the type tag that had been previously applied to the JavaScript object.

```
typedef struct {
  uint64_t lower;
  uint64_t upper;
} napi_type_tag;
```

`napi_async_cleanup_hook_handle`

An opaque value returned by `napi_add_async_cleanup_hook`. It must be passed to `napi_remove_async_cleanup_hook` when the chain of asynchronous cleanup events completes.

Node-API callback types

`napi_callback_info`

Opaque datatype that is passed to a callback function. It can be used for getting additional information about the context in which the callback was invoked.

`napi_callback`

Function pointer type for user-provided native functions which are to be exposed to JavaScript via Node-API. Callback functions should satisfy the following signature:

```
typedef napi_value (*napi_callback)(napi_env, napi_callback_info);
```

Unless for reasons discussed in Object Lifetime Management, creating a handle and/or callback scope inside a `napi_callback` is not necessary.

`napi_finalize`

Function pointer type for add-on provided functions that allow the user to be notified when externally-owned data is ready to be cleaned up because the object with which it was associated with, has been garbage-collected. The user must provide a function satisfying the following signature which would get called upon the object's collection. Currently, `napi_finalize` can be used for finding out when objects that have external data are collected.

```
typedef void (*napi_finalize)(napi_env env,
                              void* finalize_data,
                              void* finalize_hint);
```

Unless for reasons discussed in Object Lifetime Management, creating a handle and/or callback scope inside the function body is not necessary.

`napi_async_execute_callback`

Function pointer used with functions that support asynchronous operations. Callback functions must satisfy the following signature:

```
typedef void (*napi_async_execute_callback)(napi_env env, void* data);
```

Implementations of this function must avoid making Node-API calls that execute JavaScript or interact with JavaScript objects. Node-API calls should be in the `napi_async_complete_callback` instead. Do not use the `napi_env` parameter as it will likely result in execution of JavaScript.

`napi_async_complete_callback`

Function pointer used with functions that support asynchronous operations. Callback functions must satisfy the following signature:

```
typedef void (*napi_async_complete_callback)(napi_env env,
                                             napi_status status,
                                             void* data);
```

Unless for reasons discussed in Object Lifetime Management, creating a handle and/or callback scope inside the function body is not necessary.

`napi_threadsafe_function_call_js`

Function pointer used with asynchronous thread-safe function calls. The callback will be called on the main thread. Its purpose is to use a data item arriving via the queue from one of the secondary threads to construct the parameters necessary for a call into JavaScript, usually via `napi_call_function`, and then make the call into JavaScript.

The data arriving from the secondary thread via the queue is given in the `data` parameter and the JavaScript function to call is given in the `js_callback` parameter.

Node-API sets up the environment prior to calling this callback, so it is sufficient to call the JavaScript function via `napi_call_function` rather than via `napi_make_callback`.

Callback functions must satisfy the following signature:

```
typedef void (*napi_threadsafe_function_call_js)(napi_env env,
                                                 napi_value js_callback,
                                                 void* context,
                                                 void* data);
```

- `[in] env`: The environment to use for API calls, or `NULL` if the thread-safe function is being torn down and `data` may need to be freed.
- `[in] js_callback`: The JavaScript function to call, or `NULL` if the thread-safe function is being torn down and `data` may need to be freed. It may also be `NULL` if the thread-safe function was created without `js_callback`.
- `[in] context`: The optional data with which the thread-safe function was created.
- `[in] data`: Data created by the secondary thread. It is the responsibility of the callback to convert this native data to JavaScript values (with Node-API functions) that can be passed as parameters when `js_callback` is invoked. This pointer is managed entirely by the threads and this callback. Thus this callback should free the data.

Unless for reasons discussed in Object Lifetime Management, creating a handle and/or callback scope inside the function body is not necessary.

`napi_async_cleanup_hook`

Function pointer used with `napi_add_async_cleanup_hook`. It will be called when the environment is being torn down.

Callback functions must satisfy the following signature:

```
typedef void (*napi_async_cleanup_hook)(napi_async_cleanup_hook_handle handle,
                                        void* data);
```

- `[in] handle`: The handle that must be passed to `napi_remove_async_cleanup_hook` after completion of the asynchronous cleanup.
- `[in] data`: The data that was passed to `napi_add_async_cleanup_hook`.

The body of the function should initiate the asynchronous cleanup actions at the end of which `handle` must be passed in a call to `napi_remove_async_cleanup_hook`.

Error handling

Node-API uses both return values and JavaScript exceptions for error handling. The following sections explain the approach for each case.

Return values

All of the Node-API functions share the same error handling pattern. The return type of all API functions is `napi_status`.

The return value will be `napi_ok` if the request was successful and no uncaught JavaScript exception was thrown. If an error occurred AND an exception was thrown, the `napi_status` value for the error will be returned. If an exception was thrown, and no error occurred, `napi_pending_exception` will be returned.

In cases where a return value other than `napi_ok` or `napi_pending_exception` is returned, `napi_is_exception_pending` must be called to check if an exception is pending. See the section on exceptions for more details.

The full set of possible `napi_status` values is defined in `napi_api_types.h`.

The `napi_status` return value provides a VM-independent representation of the error which occurred. In some cases it is useful to be able to get more detailed information, including a string representing the error as well as VM (engine)-specific information.

In order to retrieve this information `napi_get_last_error_info` is provided which returns a `napi_extended_error_info` structure. The format of the `napi_extended_error_info` structure is as follows:

```
typedef struct napi_extended_error_info {
  const char* error_message;
  void* engine_reserved;
  uint32_t engine_error_code;
  napi_status error_code;
};
```

- `error_message`: Textual representation of the error that occurred.
- `engine_reserved`: Opaque handle reserved for engine use only.
- `engine_error_code`: VM specific error code.
- `error_code`: Node-API status code for the last error.

`napi_get_last_error_info` returns the information for the last Node-API call that was made.

Do not rely on the content or format of any of the extended information as it is not subject to SemVer and may change at any time. It is intended only for logging purposes.

`napi_get_last_error_info`

```
napi_status
napi_get_last_error_info(napi_env env,
                         const napi_extended_error_info** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: The `napi_extended_error_info` structure with more information about the error.

Returns `napi_ok` if the API succeeded.

This API retrieves a `napi_extended_error_info` structure with information about the last error that occurred.

The content of the `napi_extended_error_info` returned is only valid up until a Node-API function is called on the same `env`. This includes a call to `napi_is_exception_pending` so it may often be necessary to make a copy of the information so that it can be used later. The pointer returned in `error_message` points to a statically-defined string so it is safe to use that pointer if you have copied it out of the `error_message` field (which will be overwritten) before another Node-API function was called.

Do not rely on the content or format of any of the extended information as it is not subject to SemVer and may change at any time. It is intended only for logging purposes.

This API can be called even if there is a pending JavaScript exception.

Exceptions

Any Node-API function call may result in a pending JavaScript exception. This is the case for any of the API functions, even those that may not cause the execution of JavaScript.

If the `napi_status` returned by a function is `napi_ok` then no exception is pending and no additional action is required. If the `napi_status` returned is anything other than `napi_ok` or `napi_pending_exception`, in order to try to recover and continue instead of simply returning immediately, `napi_is_exception_pending` must be called in order to determine if an exception is pending or not.

In many cases when a Node-API function is called and an exception is already pending, the function will return immediately with a `napi_status` of `napi_pending_exception`. However, this is not the case for all functions. Node-API allows a subset of the functions to be called to allow for some minimal cleanup before returning to JavaScript. In that case, `napi_status` will reflect the status for the function. It will not reflect previous pending exceptions. To avoid confusion, check the error status after every function call.

When an exception is pending one of two approaches can be employed.

The first approach is to do any appropriate cleanup and then return so that execution will return to JavaScript. As part of the transition back to JavaScript, the exception will be thrown at the point in the JavaScript code where the native method was invoked. The behavior of most Node-API calls is unspecified while an exception is pending, and many will simply return `napi_pending_exception`, so do as little as possible and then return to JavaScript where the exception can be handled.

The second approach is to try to handle the exception. There will be cases where the native code can catch the exception, take the appropriate action, and then continue. This is only recommended in specific cases where it is known that the exception can be safely handled. In these cases `napi_get_and_clear_last_exception` can be used to get and clear the exception. On success, result will contain the handle to the last JavaScript `Object` thrown. If it is determined, after retrieving the exception, the exception cannot be handled after all it can be re-thrown it with `napi_throw` where error is the JavaScript value to be thrown.

The following utility functions are also available in case native code needs to throw an exception or determine if a `napi_value` is an instance of a JavaScript `Error` object: `napi_throw_error`, `napi_throw_type_error`, `napi_throw_range_error`, `node_api_throw_syntax_error` and `napi_is_error`.

The following utility functions are also available in case native code needs to create an `Error` object: `napi_create_error`, `napi_create_type_error`, `napi_create_range_error` and `node_api_create_syntax_error`, where result is the `napi_value` that refers to the newly created JavaScript `Error` object.

The Node.js project is adding error codes to all of the errors generated internally. The goal is for applications to use these error codes for all error checking. The associated error messages will remain, but will only be meant to be used

for logging and display with the expectation that the message can change without SemVer applying. In order to support this model with Node-API, both in internal functionality and for module specific functionality (as its good practice), the `throw_` and `create_` functions take an optional code parameter which is the string for the code to be added to the error object. If the optional parameter is `NULL` then no code will be associated with the error. If a code is provided, the name associated with the error is also updated to be:

```
originalName [code]
```

where `originalName` is the original name associated with the error and `code` is the code that was provided. For example, if the code is `'ERR_ERROR_1'` and a `TypeError` is being created the name will be:

```
TypeError [ERR_ERROR_1]
```

`napi_throw`

```
NAPI_EXTERN napi_status napi_throw(napi_env env, napi_value error);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] error` : The JavaScript value to be thrown.

Returns `napi_ok` if the API succeeded.

This API throws the JavaScript value provided.

`napi_throw_error`

```
NAPI_EXTERN napi_status napi_throw_error(napi_env env,
                                         const char* code,
                                         const char* msg);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional error code to be set on the error.
- `[in] msg` : C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `Error` with the text provided.

`napi_throw_type_error`

```
NAPI_EXTERN napi_status napi_throw_type_error(napi_env env,
                                              const char* code,
                                              const char* msg);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional error code to be set on the error.
- `[in] msg` : C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `TypeError` with the text provided.

`napi_throw_range_error`

```
NAPI_EXTERN napi_status napi_throw_range_error(napi_env env,
                                               const char* code,
                                               const char* msg);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional error code to be set on the error.
- `[in] msg` : C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `RangeError` with the text provided.

`node_api_throw_syntax_error`

```
NAPI_EXTERN napi_status node_api_throw_syntax_error(napi_env env,
                                                    const char* code,
                                                    const char* msg);

* `[in] env`: The environment that the API is invoked under.
* `[in] code`: Optional error code to be set on the error.
* `[in] msg`: C string representing the text to be associated with the error.

Returns `napi_ok` if the API succeeded.

This API throws a JavaScript `SyntaxError` with the text provided.

#### `napi_is_error`

<!-- YAML
added: v8.0.0
napiVersion: 1
-->

```c
NAPI_EXTERN napi_status napi_is_error(napi_env env,
                                      napi_value value,
                                      bool* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The `napi_value` to be checked.
- `[out] result` : Boolean value that is set to true if `napi_value` represents an error, false otherwise.

Returns `napi_ok` if the API succeeded.

This API queries a `napi_value` to check if it represents an error object.

`napi_create_error`
```

```
NAPI_EXTERN napi_status napi_create_error(napi_env env,
                                          napi_value code,
                                          napi_value msg,
                                          napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg` : `napi_value` that references a JavaScript `string` to be used as the message for the `Error` .
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `Error` with the text provided.

`napi_create_type_error`

```
NAPI_EXTERN napi_status napi_create_type_error(napi_env env,
                                               napi_value code,
                                               napi_value msg,
                                               napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg` : `napi_value` that references a JavaScript `string` to be used as the message for the `Error` .
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `TypeError` with the text provided.

`napi_create_range_error`

```
NAPI_EXTERN napi_status napi_create_range_error(napi_env env,
                                                napi_value code,
                                                napi_value msg,
                                                napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg` : `napi_value` that references a JavaScript `string` to be used as the message for the `Error` .
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `RangeError` with the text provided.

`node_api_create_syntax_error`

```
NAPI_EXTERN napi_status node_api_create_syntax_error(napi_env env,
                                                     napi_value code,
                                                     napi_value msg,
                                                     napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] code` : Optional `napi_value` with the string for the error code to be associated with the error.
- `[in] msg` : `napi_value` that references a JavaScript `string` to be used as the message for the `Error` .
- `[out] result` : `napi_value` representing the error created.

Returns `napi_ok` if the API succeeded.

This API returns a JavaScript `SyntaxError` with the text provided.

`napi_get_and_clear_last_exception`

```
napi_status napi_get_and_clear_last_exception(napi_env env,
                                              napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : The exception if one is pending, `NULL` otherwise.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

`napi_is_exception_pending`

```
napi_status napi_is_exception_pending(napi_env env, bool* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : Boolean value that is set to true if an exception is pending.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

`napi_fatal_exception`

```
napi_status napi_fatal_exception(napi_env env, napi_value err);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] err` : The error that is passed to `'uncaughtException'` .

Trigger an `'uncaughtException'` in JavaScript. Useful if an async callback throws an exception with no way to recover.

Fatal errors

In the event of an unrecoverable error in a native module, a fatal error can be thrown to immediately terminate the process.

`napi_fatal_error`

```
NAPI_NO_RETURN void napi_fatal_error(const char* location,
                                     size_t location_len,
                                     const char* message,
                                     size_t message_len);
```

- `[in] location` : Optional location at which the error occurred.
- `[in] location_len` : The length of the location in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] message` : The message associated with the error.
- `[in] message_len` : The length of the message in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.

The function call does not return, the process will be terminated.

This API can be called even if there is a pending JavaScript exception.

Object lifetime management

As Node-API calls are made, handles to objects in the heap for the underlying VM may be returned as `napi_values` . These handles must hold the objects 'live' until they are no longer required by the native code, otherwise the objects could be collected before the native code was finished using them.

As object handles are returned they are associated with a 'scope'. The lifespan for the default scope is tied to the lifespan of the native method call. The result is that, by default, handles remain valid and the objects associated with these handles will be held live for the lifespan of the native method call.

In many cases, however, it is necessary that the handles remain valid for either a shorter or longer lifespan than that of the native method. The sections which follow describe the Node-API functions that can be used to change the handle lifespan from the default.

Making handle lifespan shorter than that of the native method

It is often necessary to make the lifespan of handles shorter than the lifespan of a native method. For example, consider a native method that has a loop which iterates through the elements in a large array:

```
for (int i = 0; i < 1000000; i++) {
  napi_value result;
  napi_status status = napi_get_element(env, object, i, &result);
  if (status != napi_ok) {
    break;
  }
  // do something with element
}
```

This would result in a large number of handles being created, consuming substantial resources. In addition, even though the native code could only use the most recent handle, all of the associated objects would also be kept alive

since they all share the same scope.

To handle this case, Node-API provides the ability to establish a new 'scope' to which newly created handles will be associated. Once those handles are no longer required, the scope can be 'closed' and any handles associated with the scope are invalidated. The methods available to open/close scopes are `napi_open_handle_scope` and `napi_close_handle_scope`.

Node-API only supports a single nested hierarchy of scopes. There is only one active scope at any time, and all new handles will be associated with that scope while it is active. Scopes must be closed in the reverse order from which they are opened. In addition, all scopes created within a native method must be closed before returning from that method.

Taking the earlier example, adding calls to `napi_open_handle_scope` and `napi_close_handle_scope` would ensure that at most a single handle is valid throughout the execution of the loop:

```
for (int i = 0; i < 1000000; i++) {
  napi_handle_scope scope;
  napi_status status = napi_open_handle_scope(env, &scope);
  if (status != napi_ok) {
    break;
  }
  napi_value result;
  status = napi_get_element(env, object, i, &result);
  if (status != napi_ok) {
    break;
  }
  // do something with element
  status = napi_close_handle_scope(env, scope);
  if (status != napi_ok) {
    break;
  }
}
```

When nesting scopes, there are cases where a handle from an inner scope needs to live beyond the lifespan of that scope. Node-API supports an 'escapable scope' in order to support this case. An escapable scope allows one handle to be 'promoted' so that it 'escapes' the current scope and the lifespan of the handle changes from the current scope to that of the outer scope.

The methods available to open/close escapable scopes are `napi_open_escapable_handle_scope` and `napi_close_escapable_handle_scope`.

The request to promote a handle is made through `napi_escape_handle` which can only be called once.

`napi_open_handle_scope`

```
NAPI_EXTERN napi_status napi_open_handle_scope(napi_env env,
                                               napi_handle_scope* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: `napi_value` representing the new scope.

Returns `napi_ok` if the API succeeded.

This API opens a new scope.

`napi_close_handle_scope`

```
NAPI_EXTERN napi_status napi_close_handle_scope(napi_env env,
                                                napi_handle_scope scope);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] scope` : `napi_value` representing the scope to be closed.

Returns `napi_ok` if the API succeeded.

This API closes the scope passed in. Scopes must be closed in the reverse order from which they were created.

This API can be called even if there is a pending JavaScript exception.

`napi_open_escapable_handle_scope`

```
NAPI_EXTERN napi_status
    napi_open_escapable_handle_scope(napi_env env,
                                     napi_handle_scope* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing the new scope.

Returns `napi_ok` if the API succeeded.

This API opens a new scope from which one object can be promoted to the outer scope.

`napi_close_escapable_handle_scope`

```
NAPI_EXTERN napi_status
    napi_close_escapable_handle_scope(napi_env env,
                                      napi_handle_scope scope);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] scope` : `napi_value` representing the scope to be closed.

Returns `napi_ok` if the API succeeded.

This API closes the scope passed in. Scopes must be closed in the reverse order from which they were created.

This API can be called even if there is a pending JavaScript exception.

`napi_escape_handle`

```
napi_status napi_escape_handle(napi_env env,
                               napi_escapable_handle_scope scope,
                               napi_value escapee,
                               napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] scope`: `napi_value` representing the current scope.
- `[in] escapee`: `napi_value` representing the JavaScript `Object` to be escaped.
- `[out] result`: `napi_value` representing the handle to the escaped `Object` in the outer scope.

Returns `napi_ok` if the API succeeded.

This API promotes the handle to the JavaScript object so that it is valid for the lifetime of the outer scope. It can only be called once per scope. If it is called more than once an error will be returned.

This API can be called even if there is a pending JavaScript exception.

References to objects with a lifespan longer than that of the native method

In some cases an addon will need to be able to create and reference objects with a lifespan longer than that of a single native method invocation. For example, to create a constructor and later use that constructor in a request to creates instances, it must be possible to reference the constructor object across many different instance creation requests. This would not be possible with a normal handle returned as a `napi_value` as described in the earlier section. The lifespan of a normal handle is managed by scopes and all scopes must be closed before the end of a native method.

Node-API provides methods to create persistent references to an object. Each persistent reference has an associated count with a value of 0 or higher. The count determines if the reference will keep the corresponding object live. References with a count of 0 do not prevent the object from being collected and are often called 'weak' references. Any count greater than 0 will prevent the object from being collected.

References can be created with an initial reference count. The count can then be modified through `napi_reference_ref` and `napi_reference_unref`. If an object is collected while the count for a reference is 0, all subsequent calls to get the object associated with the reference `napi_get_reference_value` will return `NULL` for the returned `napi_value`. An attempt to call `napi_reference_ref` for a reference whose object has been collected results in an error.

References must be deleted once they are no longer required by the addon. When a reference is deleted, it will no longer prevent the corresponding object from being collected. Failure to delete a persistent reference results in a 'memory leak' with both the native memory for the persistent reference and the corresponding object on the heap being retained forever.

There can be multiple persistent references created which refer to the same object, each of which will either keep the object live or not based on its individual count. Multiple persistent references to the same object can result in unexpectedly keeping alive native memory. The native structures for a persistent reference must be kept alive until finalizers for the referenced object are executed. If a new persistent reference is created for the same object, the finalizers for that object will not be run and the native memory pointed by the earlier persistent reference will not be freed. This can be avoided by calling `napi_delete_reference` in addition to `napi_reference_unref` when possible.

`napi_create_reference`

```
NAPI_EXTERN napi_status napi_create_reference(napi_env env,
                                              napi_value value,
                                              uint32_t initial_refcount,
                                              napi_ref* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing the `Object` to which we want a reference.
- `[in] initial_refcount` : Initial reference count for the new reference.
- `[out] result` : `napi_ref` pointing to the new reference.

Returns `napi_ok` if the API succeeded.

This API creates a new reference with the specified reference count to the `Object` passed in.

`napi_delete_reference`

```
NAPI_EXTERN napi_status napi_delete_reference(napi_env env, napi_ref ref);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` to be deleted.

Returns `napi_ok` if the API succeeded.

This API deletes the reference passed in.

This API can be called even if there is a pending JavaScript exception.

`napi_reference_ref`

```
NAPI_EXTERN napi_status napi_reference_ref(napi_env env,
                                           napi_ref ref,
                                           uint32_t* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which the reference count will be incremented.
- `[out] result` : The new reference count.

Returns `napi_ok` if the API succeeded.

This API increments the reference count for the reference passed in and returns the resulting reference count.

`napi_reference_unref`

```
NAPI_EXTERN napi_status napi_reference_unref(napi_env env,
                                             napi_ref ref,
                                             uint32_t* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which the reference count will be decremented.
- `[out] result` : The new reference count.

Returns `napi_ok` if the API succeeded.

This API decrements the reference count for the reference passed in and returns the resulting reference count.

`napi_get_reference_value`

```
NAPI_EXTERN napi_status napi_get_reference_value(napi_env env,
                                                 napi_ref ref,
                                                 napi_value* result);
```

the `napi_value passed` in or out of these methods is a handle to the object to which the reference is related.

- `[in] env` : The environment that the API is invoked under.
- `[in] ref` : `napi_ref` for which we requesting the corresponding `Object` .
- `[out] result` : The `napi_value` for the `Object` referenced by the `napi_ref` .

Returns `napi_ok` if the API succeeded.

If still valid, this API returns the `napi_value` representing the JavaScript `Object` associated with the `napi_ref` . Otherwise, result will be `NULL` .

Cleanup on exit of the current Node.js instance

While a Node.js process typically releases all its resources when exiting, embedders of Node.js, or future Worker support, may require addons to register clean-up hooks that will be run once the current Node.js instance exits.

Node-API provides functions for registering and un-registering such callbacks. When those callbacks are run, all resources that are being held by the addon should be freed up.

`napi_add_env_cleanup_hook`

```
NODE_EXTERN napi_status napi_add_env_cleanup_hook(napi_env env,
                                                  void (*fun)(void* arg),
                                                  void* arg);
```

Registers `fun` as a function to be run with the `arg` parameter once the current Node.js environment exits.

A function can safely be specified multiple times with different `arg` values. In that case, it will be called multiple times as well. Providing the same `fun` and `arg` values multiple times is not allowed and will lead the process to abort.

The hooks will be called in reverse order, i.e. the most recently added one will be called first.

Removing this hook can be done by using `napi_remove_env_cleanup_hook` . Typically, that happens when the resource for which this hook was added is being torn down anyway.

For asynchronous cleanup, `napi_add_async_cleanup_hook` is available.

`napi_remove_env_cleanup_hook`

```
NAPI_EXTERN napi_status napi_remove_env_cleanup_hook(napi_env env,
                                                     void (*fun)(void* arg),
                                                     void* arg);
```

Unregisters `fun` as a function to be run with the `arg` parameter once the current Node.js environment exits. Both the argument and the function value need to be exact matches.

The function must have originally been registered with `napi_add_env_cleanup_hook`, otherwise the process will abort.

`napi_add_async_cleanup_hook`

```
NAPI_EXTERN napi_status napi_add_async_cleanup_hook(
    napi_env env,
    napi_async_cleanup_hook hook,
    void* arg,
    napi_async_cleanup_hook_handle* remove_handle);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] hook`: The function pointer to call at environment teardown.
- `[in] arg`: The pointer to pass to `hook` when it gets called.
- `[out] remove_handle`: Optional handle that refers to the asynchronous cleanup hook.

Registers `hook`, which is a function of type `napi_async_cleanup_hook`, as a function to be run with the `remove_handle` and `arg` parameters once the current Node.js environment exits.

Unlike `napi_add_env_cleanup_hook`, the hook is allowed to be asynchronous.

Otherwise, behavior generally matches that of `napi_add_env_cleanup_hook`.

If `remove_handle` is not `NULL`, an opaque value will be stored in it that must later be passed to `napi_remove_async_cleanup_hook`, regardless of whether the hook has already been invoked. Typically, that happens when the resource for which this hook was added is being torn down anyway.

`napi_remove_async_cleanup_hook`

```
NAPI_EXTERN napi_status napi_remove_async_cleanup_hook(
    napi_async_cleanup_hook_handle remove_handle);
```

- `[in] remove_handle`: The handle to an asynchronous cleanup hook that was created with `napi_add_async_cleanup_hook`.

Unregisters the cleanup hook corresponding to `remove_handle`. This will prevent the hook from being executed, unless it has already started executing. This must be called on any `napi_async_cleanup_hook_handle` value obtained from `napi_add_async_cleanup_hook`.

Module registration

Node-API modules are registered in a manner similar to other modules except that instead of using the `NODE_MODULE` macro the following is used:

```
NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)
```

The next difference is the signature for the `Init` method. For a Node-API module it is as follows:

```
napi_value Init(napi_env env, napi_value exports);
```

The return value from `Init` is treated as the `exports` object for the module. The `Init` method is passed an empty object via the `exports` parameter as a convenience. If `Init` returns `NULL`, the parameter passed as `exports` is exported by the module. Node-API modules cannot modify the `module` object but can specify anything as the `exports` property of the module.

To add the method `hello` as a function so that it can be called as a method provided by the addon:

```
napi_value Init(napi_env env, napi_value exports) {
  napi_status status;
  napi_property_descriptor desc = {
    "hello",
    NULL,
    Method,
    NULL,
    NULL,
    NULL,
    napi_writable | napi_enumerable | napi_configurable,
    NULL
  };
  status = napi_define_properties(env, exports, 1, &desc);
  if (status != napi_ok) return NULL;
  return exports;
}
```

To set a function to be returned by the `require()` for the addon:

```
napi_value Init(napi_env env, napi_value exports) {
  napi_value method;
  napi_status status;
  status = napi_create_function(env, "exports", NAPI_AUTO_LENGTH, Method, NULL,
&method);
  if (status != napi_ok) return NULL;
  return method;
}
```

To define a class so that new instances can be created (often used with Object wrap):

```
// NOTE: partial example, not all referenced code is included
napi_value Init(napi_env env, napi_value exports) {
  napi_status status;
  napi_property_descriptor properties[] = {
    { "value", NULL, NULL, GetValue, SetValue, NULL, napi_writable |
napi_configurable, NULL },
    DECLARE_NAPI_METHOD("plusOne", PlusOne),
    DECLARE_NAPI_METHOD("multiply", Multiply),
  };

  napi_value cons;
  status =
      napi_define_class(env, "MyObject", New, NULL, 3, properties, &cons);
```

```
  if (status != napi_ok) return NULL;

  status = napi_create_reference(env, cons, 1, &constructor);
  if (status != napi_ok) return NULL;

  status = napi_set_named_property(env, exports, "MyObject", cons);
  if (status != napi_ok) return NULL;

  return exports;
}
```

You can also use the `NAPI_MODULE_INIT` macro, which acts as a shorthand for `NAPI_MODULE` and defining an `Init` function:

```
NAPI_MODULE_INIT() {
  napi_value answer;
  napi_status result;

  status = napi_create_int64(env, 42, &answer);
  if (status != napi_ok) return NULL;

  status = napi_set_named_property(env, exports, "answer", answer);
  if (status != napi_ok) return NULL;

  return exports;
}
```

All Node-API addons are context-aware, meaning they may be loaded multiple times. There are a few design considerations when declaring such a module. The documentation on context-aware addons provides more details.

The variables `env` and `exports` will be available inside the function body following the macro invocation.

For more details on setting properties on objects, see the section on Working with JavaScript properties.

For more details on building addon modules in general, refer to the existing API.

Working with JavaScript values

Node-API exposes a set of APIs to create all types of JavaScript values. Some of these types are documented under Section 6 of the ECMAScript Language Specification.

Fundamentally, these APIs are used to do one of the following:

1. Create a new JavaScript object
2. Convert from a primitive C type to a Node-API value
3. Convert from Node-API value to a primitive C type
4. Get global instances including `undefined` and `null`

Node-API values are represented by the type `napi_value`. Any Node-API call that requires a JavaScript value takes in a `napi_value`. In some cases, the API does check the type of the `napi_value` up-front. However, for better performance, it's better for the caller to make sure that the `napi_value` in question is of the JavaScript type expected by the API.

Enum types

`napi_key_collection_mode`

```
typedef enum {
  napi_key_include_prototypes,
  napi_key_own_only
} napi_key_collection_mode;
```

Describes the `Keys/Properties` filter enums:

`napi_key_collection_mode` limits the range of collected properties.

`napi_key_own_only` limits the collected properties to the given object only.
`napi_key_include_prototypes` will include all keys of the objects's prototype chain as well.

`napi_key_filter`

```
typedef enum {
  napi_key_all_properties = 0,
  napi_key_writable = 1,
  napi_key_enumerable = 1 << 1,
  napi_key_configurable = 1 << 2,
  napi_key_skip_strings = 1 << 3,
  napi_key_skip_symbols = 1 << 4
} napi_key_filter;
```

Property filter bits. They can be or'ed to build a composite filter.

`napi_key_conversion`

```
typedef enum {
  napi_key_keep_numbers,
  napi_key_numbers_to_strings
} napi_key_conversion;
```

`napi_key_numbers_to_strings` will convert integer indices to strings. `napi_key_keep_numbers` will return numbers for integer indices.

`napi_valuetype`

```
typedef enum {
  // ES6 types (corresponds to typeof)
  napi_undefined,
  napi_null,
  napi_boolean,
  napi_number,
  napi_string,
  napi_symbol,
```

```
  napi_object,
  napi_function,
  napi_external,
  napi_bigint,
} napi_valuetype;
```

Describes the type of a `napi_value`. This generally corresponds to the types described in Section 6.1 of the ECMAScript Language Specification. In addition to types in that section, `napi_valuetype` can also represent `Function`s and `Object`s with external data.

A JavaScript value of type `napi_external` appears in JavaScript as a plain object such that no properties can be set on it, and no prototype.

`napi_typedarray_type`

```
typedef enum {
  napi_int8_array,
  napi_uint8_array,
  napi_uint8_clamped_array,
  napi_int16_array,
  napi_uint16_array,
  napi_int32_array,
  napi_uint32_array,
  napi_float32_array,
  napi_float64_array,
  napi_bigint64_array,
  napi_biguint64_array,
} napi_typedarray_type;
```

This represents the underlying binary scalar datatype of the `TypedArray`. Elements of this enum correspond to Section 22.2 of the ECMAScript Language Specification.

Object creation functions

`napi_create_array`

```
napi_status napi_create_array(napi_env env, napi_value* result)
```

- `[in] env`: The environment that the Node-API call is invoked under.
- `[out] result`: A `napi_value` representing a JavaScript `Array`.

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `Array` type. JavaScript arrays are described in Section 22.1 of the ECMAScript Language Specification.

`napi_create_array_with_length`

```
napi_status napi_create_array_with_length(napi_env env,
                                          size_t length,
```

```
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : The initial length of the `Array` .
- `[out] result` : A `napi_value` representing a JavaScript `Array` .

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `Array` type. The `Array` 's length property is set to the passed-in length parameter. However, the underlying buffer is not guaranteed to be pre-allocated by the VM when the array is created. That behavior is left to the underlying VM implementation. If the buffer must be a contiguous block of memory that can be directly read and/or written via C, consider using `napi_create_external_arraybuffer` .

JavaScript arrays are described in Section 22.1 of the ECMAScript Language Specification.

`napi_create_arraybuffer`

```
napi_status napi_create_arraybuffer(napi_env env,
                                    size_t byte_length,
                                    void** data,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : The length in bytes of the array buffer to create.
- `[out] data` : Pointer to the underlying byte buffer of the `ArrayBuffer` . `data` can optionally be ignored by passing `NULL` .
- `[out] result` : A `napi_value` representing a JavaScript `ArrayBuffer` .

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `ArrayBuffer` . `ArrayBuffer` s are used to represent fixed-length binary data buffers. They are normally used as a backing-buffer for `TypedArray` objects. The `ArrayBuffer` allocated will have an underlying byte buffer whose size is determined by the `length` parameter that's passed in. The underlying buffer is optionally returned back to the caller in case the caller wants to directly manipulate the buffer. This buffer can only be written to directly from native code. To write to this buffer from JavaScript, a typed array or `DataView` object would need to be created.

JavaScript `ArrayBuffer` objects are described in Section 24.1 of the ECMAScript Language Specification.

`napi_create_buffer`

```
napi_status napi_create_buffer(napi_env env,
                               size_t size,
                               void** data,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] size` : Size in bytes of the underlying buffer.
- `[out] data` : Raw pointer to the underlying buffer. `data` can optionally be ignored by passing `NULL` .

- `[out] result` : A `napi_value` representing a `node::Buffer` .

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object. While this is still a fully-supported data structure, in most cases using a `TypedArray` will suffice.

`napi_create_buffer_copy`

```
napi_status napi_create_buffer_copy(napi_env env,
                                    size_t length,
                                    const void* data,
                                    void** result_data,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] size` : Size in bytes of the input buffer (should be the same as the size of the new buffer).
- `[in] data` : Raw pointer to the underlying buffer to copy from.
- `[out] result_data` : Pointer to the new `Buffer` 's underlying data buffer. `result_data` can optionally be ignored by passing `NULL` .
- `[out] result` : A `napi_value` representing a `node::Buffer` .

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object and initializes it with data copied from the passed-in buffer. While this is still a fully-supported data structure, in most cases using a `TypedArray` will suffice.

`napi_create_date`

```
napi_status napi_create_date(napi_env env,
                             double time,
                             napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] time` : ECMAScript time value in milliseconds since 01 January, 1970 UTC.
- `[out] result` : A `napi_value` representing a JavaScript `Date` .

Returns `napi_ok` if the API succeeded.

This API does not observe leap seconds; they are ignored, as ECMAScript aligns with POSIX time specification.

This API allocates a JavaScript `Date` object.

JavaScript `Date` objects are described in Section 20.3 of the ECMAScript Language Specification.

`napi_create_external`

```
napi_status napi_create_external(napi_env env,
                                 void* data,
                                 napi_finalize finalize_cb,
```

```
                                    void* finalize_hint,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] data` : Raw pointer to the external data.
- `[in] finalize_cb` : Optional callback to call when the external value is being collected. `napi_finalize` provides more details.
- `[in] finalize_hint` : Optional hint to pass to the finalize callback during collection.
- `[out] result` : A `napi_value` representing an external value.

Returns `napi_ok` if the API succeeded.

This API allocates a JavaScript value with external data attached to it. This is used to pass external data through JavaScript code, so it can be retrieved later by native code using `napi_get_value_external` .

The API adds a `napi_finalize` callback which will be called when the JavaScript object just created is ready for garbage collection. It is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()` ,
- nor can it be removed later using `napi_remove_wrap()` , and
- the object created by the API can be used with `napi_wrap()` .

The created value is not an object, and therefore does not support additional properties. It is considered a distinct value type: calling `napi_typeof()` with an external value yields `napi_external` .

`napi_create_external_arraybuffer`

```
napi_status
napi_create_external_arraybuffer(napi_env env,
                                 void* external_data,
                                 size_t byte_length,
                                 napi_finalize finalize_cb,
                                 void* finalize_hint,
                                 napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] external_data` : Pointer to the underlying byte buffer of the `ArrayBuffer` .
- `[in] byte_length` : The length in bytes of the underlying buffer.
- `[in] finalize_cb` : Optional callback to call when the `ArrayBuffer` is being collected. `napi_finalize` provides more details.
- `[in] finalize_hint` : Optional hint to pass to the finalize callback during collection.
- `[out] result` : A `napi_value` representing a JavaScript `ArrayBuffer` .

Returns `napi_ok` if the API succeeded.

This API returns a Node-API value corresponding to a JavaScript `ArrayBuffer` . The underlying byte buffer of the `ArrayBuffer` is externally allocated and managed. The caller must ensure that the byte buffer remains valid until the finalize callback is called.

The API adds a `napi_finalize` callback which will be called when the JavaScript object just created is ready for garbage collection. It is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()`,
- nor can it be removed later using `napi_remove_wrap()`, and
- the object created by the API can be used with `napi_wrap()`.

JavaScript `ArrayBuffer`s are described in Section 24.1 of the ECMAScript Language Specification.

`napi_create_external_buffer`

```
napi_status napi_create_external_buffer(napi_env env,
                                        size_t length,
                                        void* data,
                                        napi_finalize finalize_cb,
                                        void* finalize_hint,
                                        napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] length`: Size in bytes of the input buffer (should be the same as the size of the new buffer).
- `[in] data`: Raw pointer to the underlying buffer to expose to JavaScript.
- `[in] finalize_cb`: Optional callback to call when the `ArrayBuffer` is being collected. `napi_finalize` provides more details.
- `[in] finalize_hint`: Optional hint to pass to the finalize callback during collection.
- `[out] result`: A `napi_value` representing a `node::Buffer`.

Returns `napi_ok` if the API succeeded.

This API allocates a `node::Buffer` object and initializes it with data backed by the passed in buffer. While this is still a fully-supported data structure, in most cases using a `TypedArray` will suffice.

The API adds a `napi_finalize` callback which will be called when the JavaScript object just created is ready for garbage collection. It is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()`,
- nor can it be removed later using `napi_remove_wrap()`, and
- the object created by the API can be used with `napi_wrap()`.

For Node.js >=4 `Buffers` are `Uint8Array`s.

`napi_create_object`

```
napi_status napi_create_object(napi_env env, napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[out] result`: A `napi_value` representing a JavaScript `Object`.

Returns `napi_ok` if the API succeeded.

This API allocates a default JavaScript `Object`. It is the equivalent of doing `new Object()` in JavaScript.

The JavaScript `Object` type is described in Section 6.1.7 of the ECMAScript Language Specification.

`napi_create_symbol`

```
napi_status napi_create_symbol(napi_env env,
                               napi_value description,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] description` : Optional `napi_value` which refers to a JavaScript `string` to be set as the description for the symbol.
- `[out] result` : A `napi_value` representing a JavaScript `symbol` .

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `symbol` value from a UTF8-encoded C string.

The JavaScript `symbol` type is described in Section 19.4 of the ECMAScript Language Specification.

`node_api_symbol_for`

> *Stability: 1 - Experimental*

```
napi_status node_api_symbol_for(napi_env env,
                                const char* utf8description,
                                size_t length,
                                napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] utf8description` : UTF-8 C string representing the text to be used as the description for the symbol.
- `[in] length` : The length of the description string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript `symbol` .

Returns `napi_ok` if the API succeeded.

This API searches in the global registry for an existing symbol with the given description. If the symbol already exists it will be returned, otherwise a new symbol will be created in the registry.

The JavaScript `symbol` type is described in Section 19.4 of the ECMAScript Language Specification.

`napi_create_typedarray`

```
napi_status napi_create_typedarray(napi_env env,
                                   napi_typedarray_type type,
                                   size_t length,
                                   napi_value arraybuffer,
                                   size_t byte_offset,
                                   napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] type` : Scalar datatype of the elements within the `TypedArray` .
- `[in] length` : Number of elements in the `TypedArray` .
- `[in] arraybuffer` : `ArrayBuffer` underlying the typed array.

- `[in] byte_offset` : The byte offset within the `ArrayBuffer` from which to start projecting the `TypedArray` .
- `[out] result` : A `napi_value` representing a JavaScript `TypedArray` .

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `TypedArray` object over an existing `ArrayBuffer` . `TypedArray` objects provide an array-like view over an underlying data buffer where each element has the same underlying binary scalar datatype.

It's required that `(length * size_of_element) + byte_offset` should be <= the size in bytes of the array passed in. If not, a `RangeError` exception is raised.

JavaScript `TypedArray` objects are described in Section 22.2 of the ECMAScript Language Specification.

`napi_create_dataview`

```
napi_status napi_create_dataview(napi_env env,
                                 size_t byte_length,
                                 napi_value arraybuffer,
                                 size_t byte_offset,
                                 napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] length` : Number of elements in the `DataView` .
- `[in] arraybuffer` : `ArrayBuffer` underlying the `DataView` .
- `[in] byte_offset` : The byte offset within the `ArrayBuffer` from which to start projecting the `DataView` .
- `[out] result` : A `napi_value` representing a JavaScript `DataView` .

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `DataView` object over an existing `ArrayBuffer` . `DataView` objects provide an array-like view over an underlying data buffer, but one which allows items of different size and type in the `ArrayBuffer` .

It is required that `byte_length + byte_offset` is less than or equal to the size in bytes of the array passed in. If not, a `RangeError` exception is raised.

JavaScript `DataView` objects are described in Section 24.3 of the ECMAScript Language Specification.

Functions to convert from C types to Node-API

`napi_create_int32`

```
napi_status napi_create_int32(napi_env env, int32_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number` .

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `int32_t` type to the JavaScript `number` type.

The JavaScript `number` type is described in Section 6.1.6 of the ECMAScript Language Specification.

`napi_create_uint32`

```
napi_status napi_create_uint32(napi_env env, uint32_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Unsigned integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number` .

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `uint32_t` type to the JavaScript `number` type.

The JavaScript `number` type is described in Section 6.1.6 of the ECMAScript Language Specification.

`napi_create_int64`

```
napi_status napi_create_int64(napi_env env, int64_t value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number` .

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `int64_t` type to the JavaScript `number` type.

The JavaScript `number` type is described in Section 6.1.6 of the ECMAScript Language Specification. Note the complete range of `int64_t` cannot be represented with full precision in JavaScript. Integer values outside the range of `Number.MIN_SAFE_INTEGER` `-(2**53 - 1)` - `Number.MAX_SAFE_INTEGER` `(2**53 - 1)` will lose precision.

`napi_create_double`

```
napi_status napi_create_double(napi_env env, double value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Double-precision value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `number` .

Returns `napi_ok` if the API succeeded.

This API is used to convert from the C `double` type to the JavaScript `number` type.

The JavaScript `number` type is described in Section 6.1.6 of the ECMAScript Language Specification.

`napi_create_bigint_int64`

```
napi_status napi_create_bigint_int64(napi_env env,
                                     int64_t value,
                                     napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `BigInt` .

Returns `napi_ok` if the API succeeded.

This API converts the C `int64_t` type to the JavaScript `BigInt` type.

`napi_create_bigint_uint64`

```
napi_status napi_create_bigint_uint64(napi_env env,
                                      uint64_t value,
                                      napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : Unsigned integer value to be represented in JavaScript.
- `[out] result` : A `napi_value` representing a JavaScript `BigInt` .

Returns `napi_ok` if the API succeeded.

This API converts the C `uint64_t` type to the JavaScript `BigInt` type.

`napi_create_bigint_words`

```
napi_status napi_create_bigint_words(napi_env env,
                                     int sign_bit,
                                     size_t word_count,
                                     const uint64_t* words,
                                     napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] sign_bit` : Determines if the resulting `BigInt` will be positive or negative.
- `[in] word_count` : The length of the `words` array.
- `[in] words` : An array of `uint64_t` little-endian 64-bit words.
- `[out] result` : A `napi_value` representing a JavaScript `BigInt` .

Returns `napi_ok` if the API succeeded.

This API converts an array of unsigned 64-bit words into a single `BigInt` value.

The resulting `BigInt` is calculated as: (–1)$^{\texttt{sign\_bit}}$ (`words[0]` × $(2^{64})^0$ + `words[1]` × $(2^{64})^1$ + ...)

`napi_create_string_latin1`

```
napi_status napi_create_string_latin1(napi_env env,
                                      const char* str,
```

```
                                    size_t length,
                                    napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] str` : Character buffer representing an ISO-8859-1-encoded string.
- `[in] length` : The length of the string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript `string` .

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `string` value from an ISO-8859-1-encoded C string. The native string is copied.

The JavaScript `string` type is described in Section 6.1.4 of the ECMAScript Language Specification.

`napi_create_string_utf16`

```
napi_status napi_create_string_utf16(napi_env env,
                                     const char16_t* str,
                                     size_t length,
                                     napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] str` : Character buffer representing a UTF16-LE-encoded string.
- `[in] length` : The length of the string in two-byte code units, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript `string` .

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `string` value from a UTF16-LE-encoded C string. The native string is copied.

The JavaScript `string` type is described in Section 6.1.4 of the ECMAScript Language Specification.

`napi_create_string_utf8`

```
napi_status napi_create_string_utf8(napi_env env,
                                    const char* str,
                                    size_t length,
                                    napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] str` : Character buffer representing a UTF8-encoded string.
- `[in] length` : The length of the string in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[out] result` : A `napi_value` representing a JavaScript `string` .

Returns `napi_ok` if the API succeeded.

This API creates a JavaScript `string` value from a UTF8-encoded C string. The native string is copied.

The JavaScript `string` type is described in Section 6.1.4 of the ECMAScript Language Specification.

Functions to convert from Node-API to C types

`napi_get_array_length`

```
napi_status napi_get_array_length(napi_env env,
                                  napi_value value,
                                  uint32_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing the JavaScript `Array` whose length is being queried.
- `[out] result` : `uint32` representing length of the array.

Returns `napi_ok` if the API succeeded.

This API returns the length of an array.

`Array` length is described in Section 22.1.4.1 of the ECMAScript Language Specification.

`napi_get_arraybuffer_info`

```
napi_status napi_get_arraybuffer_info(napi_env env,
                                      napi_value arraybuffer,
                                      void** data,
                                      size_t* byte_length)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] arraybuffer` : `napi_value` representing the `ArrayBuffer` being queried.
- `[out] data` : The underlying data buffer of the `ArrayBuffer` . If byte_length is `0` , this may be `NULL` or any other pointer value.
- `[out] byte_length` : Length in bytes of the underlying data buffer.

Returns `napi_ok` if the API succeeded.

This API is used to retrieve the underlying data buffer of an `ArrayBuffer` and its length.

WARNING: Use caution while using this API. The lifetime of the underlying data buffer is managed by the `ArrayBuffer` even after it's returned. A possible safe way to use this API is in conjunction with `napi_create_reference` , which can be used to guarantee control over the lifetime of the `ArrayBuffer` . It's also safe to use the returned data buffer within the same callback as long as there are no calls to other APIs that might trigger a GC.

`napi_get_buffer_info`

```
napi_status napi_get_buffer_info(napi_env env,
                                 napi_value value,
                                 void** data,
                                 size_t* length)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing the `node::Buffer` being queried.

- `[out] data` : The underlying data buffer of the `node::Buffer` . If length is `0` , this may be `NULL` or any other pointer value.
- `[out] length` : Length in bytes of the underlying data buffer.

Returns `napi_ok` if the API succeeded.

This API is used to retrieve the underlying data buffer of a `node::Buffer` and its length.

Warning: Use caution while using this API since the underlying data buffer's lifetime is not guaranteed if it's managed by the VM.

`napi_get_prototype`

```
napi_status napi_get_prototype(napi_env env,
                               napi_value object,
                               napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] object` : `napi_value` representing JavaScript `Object` whose prototype to return. This returns the equivalent of `Object.getPrototypeOf` (which is not the same as the function's `prototype` property).
- `[out] result` : `napi_value` representing prototype of the given object.

Returns `napi_ok` if the API succeeded.

`napi_get_typedarray_info`

```
napi_status napi_get_typedarray_info(napi_env env,
                                     napi_value typedarray,
                                     napi_typedarray_type* type,
                                     size_t* length,
                                     void** data,
                                     napi_value* arraybuffer,
                                     size_t* byte_offset)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] typedarray` : `napi_value` representing the `TypedArray` whose properties to query.
- `[out] type` : Scalar datatype of the elements within the `TypedArray` .
- `[out] length` : The number of elements in the `TypedArray` .
- `[out] data` : The data buffer underlying the `TypedArray` adjusted by the `byte_offset` value so that it points to the first element in the `TypedArray` . If the length of the array is `0` , this may be `NULL` or any other pointer value.
- `[out] arraybuffer` : The `ArrayBuffer` underlying the `TypedArray` .
- `[out] byte_offset` : The byte offset within the underlying native array at which the first element of the arrays is located. The value for the data parameter has already been adjusted so that data points to the first element in the array. Therefore, the first byte of the native array would be at `data - byte_offset` .

Returns `napi_ok` if the API succeeded.

This API returns various properties of a typed array.

Any of the out parameters may be `NULL` if that property is unneeded.

Warning: Use caution while using this API since the underlying data buffer is managed by the VM.

`napi_get_dataview_info`

```
napi_status napi_get_dataview_info(napi_env env,
                                   napi_value dataview,
                                   size_t* byte_length,
                                   void** data,
                                   napi_value* arraybuffer,
                                   size_t* byte_offset)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] dataview`: `napi_value` representing the `DataView` whose properties to query.
- `[out] byte_length`: Number of bytes in the `DataView`.
- `[out] data`: The data buffer underlying the `DataView`. If byte_length is `0`, this may be `NULL` or any other pointer value.
- `[out] arraybuffer`: `ArrayBuffer` underlying the `DataView`.
- `[out] byte_offset`: The byte offset within the data buffer from which to start projecting the `DataView`.

Returns `napi_ok` if the API succeeded.

Any of the out parameters may be `NULL` if that property is unneeded.

This API returns various properties of a `DataView`.

`napi_get_date_value`

```
napi_status napi_get_date_value(napi_env env,
                                napi_value value,
                                double* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: `napi_value` representing a JavaScript `Date`.
- `[out] result`: Time value as a `double` represented as milliseconds since midnight at the beginning of 01 January, 1970 UTC.

This API does not observe leap seconds; they are ignored, as ECMAScript aligns with POSIX time specification.

Returns `napi_ok` if the API succeeded. If a non-date `napi_value` is passed in it returns `napi_date_expected`.

This API returns the C double primitive of time value for the given JavaScript `Date`.

`napi_get_value_bool`

```
napi_status napi_get_value_bool(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript `Boolean` .
- `[out] result` : C boolean primitive equivalent of the given JavaScript `Boolean` .

Returns `napi_ok` if the API succeeded. If a non-boolean `napi_value` is passed in it returns `napi_boolean_expected` .

This API returns the C boolean primitive equivalent of the given JavaScript `Boolean` .

`napi_get_value_double`

```
napi_status napi_get_value_double(napi_env env,
                                  napi_value value,
                                  double* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript `number` .
- `[out] result` : C double primitive equivalent of the given JavaScript `number` .

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected` .

This API returns the C double primitive equivalent of the given JavaScript `number` .

`napi_get_value_bigint_int64`

```
napi_status napi_get_value_bigint_int64(napi_env env,
                                        napi_value value,
                                        int64_t* result,
                                        bool* lossless);
```

- `[in] env` : The environment that the API is invoked under
- `[in] value` : `napi_value` representing JavaScript `BigInt` .
- `[out] result` : C `int64_t` primitive equivalent of the given JavaScript `BigInt` .
- `[out] lossless` : Indicates whether the `BigInt` value was converted losslessly.

Returns `napi_ok` if the API succeeded. If a non- `BigInt` is passed in it returns `napi_bigint_expected` .

This API returns the C `int64_t` primitive equivalent of the given JavaScript `BigInt` . If needed it will truncate the value, setting `lossless` to `false` .

`napi_get_value_bigint_uint64`

```
napi_status napi_get_value_bigint_uint64(napi_env env,
                                         napi_value value,
                                         uint64_t* result,
                                         bool* lossless);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript `BigInt` .

- `[out] result` : C `uint64_t` primitive equivalent of the given JavaScript `BigInt` .
- `[out] lossless` : Indicates whether the `BigInt` value was converted losslessly.

Returns `napi_ok` if the API succeeded. If a non- `BigInt` is passed in it returns `napi_bigint_expected` .

This API returns the C `uint64_t` primitive equivalent of the given JavaScript `BigInt` . If needed it will truncate the value, setting `lossless` to `false` .

`napi_get_value_bigint_words`

```
napi_status napi_get_value_bigint_words(napi_env env,
                                        napi_value value,
                                        int* sign_bit,
                                        size_t* word_count,
                                        uint64_t* words);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript `BigInt` .
- `[out] sign_bit` : Integer representing if the JavaScript `BigInt` is positive or negative.
- `[in/out] word_count` : Must be initialized to the length of the `words` array. Upon return, it will be set to the actual number of words that would be needed to store this `BigInt` .
- `[out] words` : Pointer to a pre-allocated 64-bit word array.

Returns `napi_ok` if the API succeeded.

This API converts a single `BigInt` value into a sign bit, 64-bit little-endian array, and the number of elements in the array. `sign_bit` and `words` may be both set to `NULL` , in order to get only `word_count` .

`napi_get_value_external`

```
napi_status napi_get_value_external(napi_env env,
                                    napi_value value,
                                    void** result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript external value.
- `[out] result` : Pointer to the data wrapped by the JavaScript external value.

Returns `napi_ok` if the API succeeded. If a non-external `napi_value` is passed in it returns `napi_invalid_arg` .

This API retrieves the external data pointer that was previously passed to `napi_create_external()` .

`napi_get_value_int32`

```
napi_status napi_get_value_int32(napi_env env,
                                 napi_value value,
                                 int32_t* result)
```

- `[in] env` : The environment that the API is invoked under.

- `[in] value` : `napi_value` representing JavaScript `number` .
- `[out] result` : C `int32` primitive equivalent of the given JavaScript `number` .

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in `napi_number_expected` .

This API returns the C `int32` primitive equivalent of the given JavaScript `number` .

If the number exceeds the range of the 32 bit integer, then the result is truncated to the equivalent of the bottom 32 bits. This can result in a large positive number becoming a negative number if the value is > 2^{31} - 1.

Non-finite number values (`NaN` , `+Infinity` , or `-Infinity`) set the result to zero.

`napi_get_value_int64`

```
napi_status napi_get_value_int64(napi_env env,
                                 napi_value value,
                                 int64_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript `number` .
- `[out] result` : C `int64` primitive equivalent of the given JavaScript `number` .

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected` .

This API returns the C `int64` primitive equivalent of the given JavaScript `number` .

`number` values outside the range of `Number.MIN_SAFE_INTEGER` `-(2**53 - 1)` - `Number.MAX_SAFE_INTEGER` `(2**53 - 1)` will lose precision.

Non-finite number values (`NaN` , `+Infinity` , or `-Infinity`) set the result to zero.

`napi_get_value_string_latin1`

```
napi_status napi_get_value_string_latin1(napi_env env,
                                         napi_value value,
                                         char* buf,
                                         size_t bufsize,
                                         size_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript string.
- `[in] buf` : Buffer to write the ISO-8859-1-encoded string into. If `NULL` is passed in, the length of the string in bytes and excluding the null terminator is returned in `result` .
- `[in] bufsize` : Size of the destination buffer. When this value is insufficient, the returned string is truncated and null-terminated.
- `[out] result` : Number of bytes copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non- `string` `napi_value` is passed in it returns `napi_string_expected` .

This API returns the ISO-8859-1-encoded string corresponding the value passed in.

`napi_get_value_string_utf8`

```
napi_status napi_get_value_string_utf8(napi_env env,
                                       napi_value value,
                                       char* buf,
                                       size_t bufsize,
                                       size_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript string.
- `[in] buf` : Buffer to write the UTF8-encoded string into. If `NULL` is passed in, the length of the string in bytes and excluding the null terminator is returned in `result` .
- `[in] bufsize` : Size of the destination buffer. When this value is insufficient, the returned string is truncated and null-terminated.
- `[out] result` : Number of bytes copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non-`string` `napi_value` is passed in it returns `napi_string_expected` .

This API returns the UTF8-encoded string corresponding the value passed in.

`napi_get_value_string_utf16`

```
napi_status napi_get_value_string_utf16(napi_env env,
                                        napi_value value,
                                        char16_t* buf,
                                        size_t bufsize,
                                        size_t* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : `napi_value` representing JavaScript string.
- `[in] buf` : Buffer to write the UTF16-LE-encoded string into. If `NULL` is passed in, the length of the string in 2-byte code units and excluding the null terminator is returned.
- `[in] bufsize` : Size of the destination buffer. When this value is insufficient, the returned string is truncated and null-terminated.
- `[out] result` : Number of 2-byte code units copied into the buffer, excluding the null terminator.

Returns `napi_ok` if the API succeeded. If a non-`string` `napi_value` is passed in it returns `napi_string_expected` .

This API returns the UTF16-encoded string corresponding the value passed in.

`napi_get_value_uint32`

```
napi_status napi_get_value_uint32(napi_env env,
                                  napi_value value,
                                  uint32_t* result)
```

- `[in] env` : The environment that the API is invoked under.

- `[in] value` : `napi_value` representing JavaScript `number` .
- `[out] result` : C primitive equivalent of the given `napi_value` as a `uint32_t` .

Returns `napi_ok` if the API succeeded. If a non-number `napi_value` is passed in it returns `napi_number_expected` .

This API returns the C primitive equivalent of the given `napi_value` as a `uint32_t` .

Functions to get global instances

`napi_get_boolean`

```
napi_status napi_get_boolean(napi_env env, bool value, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The value of the boolean to retrieve.
- `[out] result` : `napi_value` representing JavaScript `Boolean` singleton to retrieve.

Returns `napi_ok` if the API succeeded.

This API is used to return the JavaScript singleton object that is used to represent the given boolean value.

`napi_get_global`

```
napi_status napi_get_global(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing JavaScript `global` object.

Returns `napi_ok` if the API succeeded.

This API returns the `global` object.

`napi_get_null`

```
napi_status napi_get_null(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing JavaScript `null` object.

Returns `napi_ok` if the API succeeded.

This API returns the `null` object.

`napi_get_undefined`

```
napi_status napi_get_undefined(napi_env env, napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : `napi_value` representing JavaScript Undefined value.

Returns `napi_ok` if the API succeeded.

This API returns the Undefined object.

Working with JavaScript values and abstract operations

Node-API exposes a set of APIs to perform some abstract operations on JavaScript values. Some of these operations are documented under Section 7 of the ECMAScript Language Specification.

These APIs support doing one of the following:

1. Coerce JavaScript values to specific JavaScript types (such as `number` or `string`).
2. Check the type of a JavaScript value.
3. Check for equality between two JavaScript values.

`napi_coerce_to_bool`

```
napi_status napi_coerce_to_bool(napi_env env,
                                napi_value value,
                                napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The JavaScript value to coerce.
- `[out] result`: `napi_value` representing the coerced JavaScript `Boolean`.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToBoolean()` as defined in Section 7.1.2 of the ECMAScript Language Specification.

`napi_coerce_to_number`

```
napi_status napi_coerce_to_number(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] value`: The JavaScript value to coerce.
- `[out] result`: `napi_value` representing the coerced JavaScript `number`.

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToNumber()` as defined in Section 7.1.3 of the ECMAScript Language Specification. This function potentially runs JS code if the passed-in value is an object.

`napi_coerce_to_object`

```
napi_status napi_coerce_to_object(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to coerce.
- `[out] result` : `napi_value` representing the coerced JavaScript `Object` .

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToObject()` as defined in Section 7.1.13 of the ECMAScript Language Specification.

`napi_coerce_to_string`

```
napi_status napi_coerce_to_string(napi_env env,
                                  napi_value value,
                                  napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to coerce.
- `[out] result` : `napi_value` representing the coerced JavaScript `string` .

Returns `napi_ok` if the API succeeded.

This API implements the abstract operation `ToString()` as defined in Section 7.1.13 of the ECMAScript Language Specification. This function potentially runs JS code if the passed-in value is an object.

`napi_typeof`

```
napi_status napi_typeof(napi_env env, napi_value value, napi_valuetype* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value whose type to query.
- `[out] result` : The type of the JavaScript value.

Returns `napi_ok` if the API succeeded.

- `napi_invalid_arg` if the type of `value` is not a known ECMAScript type and `value` is not an External value.

This API represents behavior similar to invoking the `typeof` Operator on the object as defined in Section 12.5.5 of the ECMAScript Language Specification. However, there are some differences:

1. It has support for detecting an External value.
2. It detects `null` as a separate type, while ECMAScript `typeof` would detect `object` .

If `value` has a type that is invalid, an error is returned.

`napi_instanceof`

```
napi_status napi_instanceof(napi_env env,
                            napi_value object,
                            napi_value constructor,
                            bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] object` : The JavaScript value to check.
- `[in] constructor` : The JavaScript function object of the constructor function to check against.
- `[out] result` : Boolean that is set to true if `object instanceof constructor` is true.

Returns `napi_ok` if the API succeeded.

This API represents invoking the `instanceof` Operator on the object as defined in Section 12.10.4 of the ECMAScript Language Specification.

`napi_is_array`

```
napi_status napi_is_array(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given object is an array.

Returns `napi_ok` if the API succeeded.

This API represents invoking the `IsArray` operation on the object as defined in Section 7.2.2 of the ECMAScript Language Specification.

`napi_is_arraybuffer`

```
napi_status napi_is_arraybuffer(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given object is an `ArrayBuffer` .

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is an array buffer.

`napi_is_buffer`

```
napi_status napi_is_buffer(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a `node::Buffer` object.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a buffer.

`napi_is_date`

```
napi_status napi_is_date(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a JavaScript `Date` object.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a date.

`napi_is_error`

```
napi_status napi_is_error(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents an `Error` object.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is an `Error` .

`napi_is_typedarray`

```
napi_status napi_is_typedarray(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a `TypedArray` .

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a typed array.

`napi_is_dataview`

```
napi_status napi_is_dataview(napi_env env, napi_value value, bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The JavaScript value to check.
- `[out] result` : Whether the given `napi_value` represents a `DataView` .

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in is a `DataView` .

`napi_strict_equals`

```
napi_status napi_strict_equals(napi_env env,
                               napi_value lhs,
                               napi_value rhs,
                               bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] lhs` : The JavaScript value to check.
- `[in] rhs` : The JavaScript value to check against.
- `[out] result` : Whether the two `napi_value` objects are equal.

Returns `napi_ok` if the API succeeded.

This API represents the invocation of the Strict Equality algorithm as defined in Section 7.2.14 of the ECMAScript Language Specification.

`napi_detach_arraybuffer`

```
napi_status napi_detach_arraybuffer(napi_env env,
                                    napi_value arraybuffer)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] arraybuffer` : The JavaScript `ArrayBuffer` to be detached.

Returns `napi_ok` if the API succeeded. If a non-detachable `ArrayBuffer` is passed in it returns `napi_detachable_arraybuffer_expected` .

Generally, an `ArrayBuffer` is non-detachable if it has been detached before. The engine may impose additional conditions on whether an `ArrayBuffer` is detachable. For example, V8 requires that the `ArrayBuffer` be external, that is, created with `napi_create_external_arraybuffer` .

This API represents the invocation of the `ArrayBuffer` detach operation as defined in Section 24.1.1.3 of the ECMAScript Language Specification.

`napi_is_detached_arraybuffer`

```
napi_status napi_is_detached_arraybuffer(napi_env env,
                                         napi_value arraybuffer,
                                         bool* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] arraybuffer` : The JavaScript `ArrayBuffer` to be checked.
- `[out] result` : Whether the `arraybuffer` is detached.

Returns `napi_ok` if the API succeeded.

The `ArrayBuffer` is considered detached if its internal data is `null` .

This API represents the invocation of the `ArrayBuffer` `IsDetachedBuffer` operation as defined in Section 24.1.1.2 of the ECMAScript Language Specification.

Working with JavaScript properties

Node-API exposes a set of APIs to get and set properties on JavaScript objects. Some of these types are documented under Section 7 of the ECMAScript Language Specification.

Properties in JavaScript are represented as a tuple of a key and a value. Fundamentally, all property keys in Node-API can be represented in one of the following forms:

- Named: a simple UTF8-encoded string
- Integer-Indexed: an index value represented by `uint32_t`
- JavaScript value: these are represented in Node-API by `napi_value`. This can be a `napi_value` representing a `string`, `number`, or `symbol`.

Node-API values are represented by the type `napi_value`. Any Node-API call that requires a JavaScript value takes in a `napi_value`. However, it's the caller's responsibility to make sure that the `napi_value` in question is of the JavaScript type expected by the API.

The APIs documented in this section provide a simple interface to get and set properties on arbitrary JavaScript objects represented by `napi_value`.

For instance, consider the following JavaScript code snippet:

```
const obj = {};
obj.myProp = 123;
```

The equivalent can be done using Node-API values with the following snippet:

```
napi_status status = napi_generic_failure;

// const obj = {}
napi_value obj, value;
status = napi_create_object(env, &obj);
if (status != napi_ok) return status;

// Create a napi_value for 123
status = napi_create_int32(env, 123, &value);
if (status != napi_ok) return status;

// obj.myProp = 123
status = napi_set_named_property(env, obj, "myProp", value);
if (status != napi_ok) return status;
```

Indexed properties can be set in a similar manner. Consider the following JavaScript snippet:

```
const arr = [];
arr[123] = 'hello';
```

The equivalent can be done using Node-API values with the following snippet:

```
napi_status status = napi_generic_failure;

// const arr = [];
napi_value arr, value;
status = napi_create_array(env, &arr);
if (status != napi_ok) return status;

// Create a napi_value for 'hello'
status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &value);
if (status != napi_ok) return status;

// arr[123] = 'hello';
status = napi_set_element(env, arr, 123, value);
if (status != napi_ok) return status;
```

Properties can be retrieved using the APIs described in this section. Consider the following JavaScript snippet:

```
const arr = [];
const value = arr[123];
```

The following is the approximate equivalent of the Node-API counterpart:

```
napi_status status = napi_generic_failure;

// const arr = []
napi_value arr, value;
status = napi_create_array(env, &arr);
if (status != napi_ok) return status;

// const value = arr[123]
status = napi_get_element(env, arr, 123, &value);
if (status != napi_ok) return status;
```

Finally, multiple properties can also be defined on an object for performance reasons. Consider the following JavaScript:

```
const obj = {};
Object.defineProperties(obj, {
  'foo': { value: 123, writable: true, configurable: true, enumerable: true },
  'bar': { value: 456, writable: true, configurable: true, enumerable: true }
});
```

The following is the approximate equivalent of the Node-API counterpart:

```
napi_status status = napi_status_generic_failure;

// const obj = {};
napi_value obj;
status = napi_create_object(env, &obj);
```

```
if (status != napi_ok) return status;

// Create napi_values for 123 and 456
napi_value fooValue, barValue;
status = napi_create_int32(env, 123, &fooValue);
if (status != napi_ok) return status;
status = napi_create_int32(env, 456, &barValue);
if (status != napi_ok) return status;

// Set the properties
napi_property_descriptor descriptors[] = {
  { "foo", NULL, NULL, NULL, NULL, fooValue, napi_writable | napi_configurable, NULL
},
  { "bar", NULL, NULL, NULL, NULL, barValue, napi_writable | napi_configurable, NULL
}
}
status = napi_define_properties(env,
                                obj,
                                sizeof(descriptors) / sizeof(descriptors[0]),
                                descriptors);
if (status != napi_ok) return status;
```

Structures

`napi_property_attributes`

```
typedef enum {
  napi_default = 0,
  napi_writable = 1 << 0,
  napi_enumerable = 1 << 1,
  napi_configurable = 1 << 2,

  // Used with napi_define_class to distinguish static properties
  // from instance properties. Ignored by napi_define_properties.
  napi_static = 1 << 10,

  // Default for class methods.
  napi_default_method = napi_writable | napi_configurable,

  // Default for object properties, like in JS obj[prop].
  napi_default_jsproperty = napi_writable |
                            napi_enumerable |
                            napi_configurable,
} napi_property_attributes;
```

`napi_property_attributes` are flags used to control the behavior of properties set on a JavaScript object. Other than `napi_static` they correspond to the attributes listed in Section 6.1.7.1 of the ECMAScript Language Specification. They can be one or more of the following bitflags:

- `napi_default` : No explicit attributes are set on the property. By default, a property is read only, not enumerable and not configurable.

- `napi_writable` : The property is writable.
- `napi_enumerable` : The property is enumerable.
- `napi_configurable` : The property is configurable as defined in Section 6.1.7.1 of the ECMAScript Language Specification.
- `napi_static` : The property will be defined as a static property on a class as opposed to an instance property, which is the default. This is used only by `napi_define_class` . It is ignored by `napi_define_properties` .
- `napi_default_method` : Like a method in a JS class, the property is configurable and writeable, but not enumerable.
- `napi_default_jsproperty` : Like a property set via assignment in JavaScript, the property is writable, enumerable, and configurable.

`napi_property_descriptor`

```
typedef struct {
  // One of utf8name or name should be NULL.
  const char* utf8name;
  napi_value name;

  napi_callback method;
  napi_callback getter;
  napi_callback setter;
  napi_value value;

  napi_property_attributes attributes;
  void* data;
} napi_property_descriptor;
```

- `utf8name` : Optional string describing the key for the property, encoded as UTF8. One of `utf8name` or `name` must be provided for the property.
- `name` : Optional `napi_value` that points to a JavaScript string or symbol to be used as the key for the property. One of `utf8name` or `name` must be provided for the property.
- `value` : The value that's retrieved by a get access of the property if the property is a data property. If this is passed in, set `getter` , `setter` , `method` and `data` to `NULL` (since these members won't be used).
- `getter` : A function to call when a get access of the property is performed. If this is passed in, set `value` and `method` to `NULL` (since these members won't be used). The given function is called implicitly by the runtime when the property is accessed from JavaScript code (or if a get on the property is performed using a Node-API call). `napi_callback` provides more details.
- `setter` : A function to call when a set access of the property is performed. If this is passed in, set `value` and `method` to `NULL` (since these members won't be used). The given function is called implicitly by the runtime when the property is set from JavaScript code (or if a set on the property is performed using a Node-API call). `napi_callback` provides more details.
- `method` : Set this to make the property descriptor object's `value` property to be a JavaScript function represented by `method` . If this is passed in, set `value` , `getter` and `setter` to `NULL` (since these members won't be used). `napi_callback` provides more details.
- `attributes` : The attributes associated with the particular property. See `napi_property_attributes` .

- `data` : The callback data passed into `method` , `getter` and `setter` if this function is invoked.

Functions

`napi_get_property_names`

```
napi_status napi_get_property_names(napi_env env,
                                    napi_value object,
                                    napi_value* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to retrieve the properties.
- `[out] result` : A `napi_value` representing an array of JavaScript values that represent the property names of the object. The API can be used to iterate over `result` using `napi_get_array_length` and `napi_get_element` .

Returns `napi_ok` if the API succeeded.

This API returns the names of the enumerable properties of `object` as an array of strings. The properties of `object` whose key is a symbol will not be included.

`napi_get_all_property_names`

```
napi_get_all_property_names(napi_env env,
                            napi_value object,
                            napi_key_collection_mode key_mode,
                            napi_key_filter key_filter,
                            napi_key_conversion key_conversion,
                            napi_value* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to retrieve the properties.
- `[in] key_mode` : Whether to retrieve prototype properties as well.
- `[in] key_filter` : Which properties to retrieve (enumerable/readable/writable).
- `[in] key_conversion` : Whether to convert numbered property keys to strings.
- `[out] result` : A `napi_value` representing an array of JavaScript values that represent the property names of the object. `napi_get_array_length` and `napi_get_element` can be used to iterate over `result` .

Returns `napi_ok` if the API succeeded.

This API returns an array containing the names of the available properties of this object.

`napi_set_property`

```
napi_status napi_set_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              napi_value value);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object on which to set the property.
- `[in] key` : The name of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This API set a property on the `Object` passed in.

`napi_get_property`

```
napi_status napi_get_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              napi_value* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to retrieve the property.
- `[in] key` : The name of the property to retrieve.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This API gets the requested property from the `Object` passed in.

`napi_has_property`

```
napi_status napi_has_property(napi_env env,
                              napi_value object,
                              napi_value key,
                              bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] key` : The name of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in has the named property.

`napi_delete_property`

```
napi_status napi_delete_property(napi_env env,
                                 napi_value object,
                                 napi_value key,
                                 bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.

- `[in] key` : The name of the property to delete.
- `[out] result` : Whether the property deletion succeeded or not. `result` can optionally be ignored by passing `NULL` .

Returns `napi_ok` if the API succeeded.

This API attempts to delete the `key` own property from `object` .

`napi_has_own_property`

```
napi_status napi_has_own_property(napi_env env,
                                  napi_value object,
                                  napi_value key,
                                  bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] key` : The name of the own property whose existence to check.
- `[out] result` : Whether the own property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API checks if the `Object` passed in has the named own property. `key` must be a `string` or a `symbol` , or an error will be thrown. Node-API will not perform any conversion between data types.

`napi_set_named_property`

```
napi_status napi_set_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    napi_value value);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object on which to set the property.
- `[in] utf8Name` : The name of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling `napi_set_property` with a `napi_value` created from the string passed in as `utf8Name` .

`napi_get_named_property`

```
napi_status napi_get_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    napi_value* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.

- `[in] object` : The object from which to retrieve the property.
- `[in] utf8Name` : The name of the property to get.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling `napi_get_property` with a `napi_value` created from the string passed in as `utf8Name` .

`napi_has_named_property`

```
napi_status napi_has_named_property(napi_env env,
                                    napi_value object,
                                    const char* utf8Name,
                                    bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] utf8Name` : The name of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This method is equivalent to calling `napi_has_property` with a `napi_value` created from the string passed in as `utf8Name` .

`napi_set_element`

```
napi_status napi_set_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             napi_value value);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to set the properties.
- `[in] index` : The index of the property to set.
- `[in] value` : The property value.

Returns `napi_ok` if the API succeeded.

This API sets an element on the `Object` passed in.

`napi_get_element`

```
napi_status napi_get_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             napi_value* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.

- `[in] object` : The object from which to retrieve the property.
- `[in] index` : The index of the property to get.
- `[out] result` : The value of the property.

Returns `napi_ok` if the API succeeded.

This API gets the element at the requested index.

`napi_has_element`

```
napi_status napi_has_element(napi_env env,
                             napi_value object,
                             uint32_t index,
                             bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] index` : The index of the property whose existence to check.
- `[out] result` : Whether the property exists on the object or not.

Returns `napi_ok` if the API succeeded.

This API returns if the `Object` passed in has an element at the requested index.

`napi_delete_element`

```
napi_status napi_delete_element(napi_env env,
                                napi_value object,
                                uint32_t index,
                                bool* result);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to query.
- `[in] index` : The index of the property to delete.
- `[out] result` : Whether the element deletion succeeded or not. `result` can optionally be ignored by passing `NULL` .

Returns `napi_ok` if the API succeeded.

This API attempts to delete the specified `index` from `object` .

`napi_define_properties`

```
napi_status napi_define_properties(napi_env env,
                                   napi_value object,
                                   size_t property_count,
                                   const napi_property_descriptor* properties);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object from which to retrieve the properties.

- `[in] property_count` : The number of elements in the `properties` array.
- `[in] properties` : The array of property descriptors.

Returns `napi_ok` if the API succeeded.

This method allows the efficient definition of multiple properties on a given object. The properties are defined using property descriptors (see `napi_property_descriptor`). Given an array of such property descriptors, this API will set the properties on the object one at a time, as defined by `DefineOwnProperty()` (described in Section 9.1.6 of the ECMA-262 specification).

`napi_object_freeze`

```
napi_status napi_object_freeze(napi_env env,
                               napi_value object);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to freeze.

Returns `napi_ok` if the API succeeded.

This method freezes a given object. This prevents new properties from being added to it, existing properties from being removed, prevents changing the enumerability, configurability, or writability of existing properties, and prevents the values of existing properties from being changed. It also prevents the object's prototype from being changed. This is described in Section 19.1.2.6 of the ECMA-262 specification.

`napi_object_seal`

```
napi_status napi_object_seal(napi_env env,
                             napi_value object);
```

- `[in] env` : The environment that the Node-API call is invoked under.
- `[in] object` : The object to seal.

Returns `napi_ok` if the API succeeded.

This method seals a given object. This prevents new properties from being added to it, as well as marking all existing properties as non-configurable. This is described in Section 19.1.2.20 of the ECMA-262 specification.

Working with JavaScript functions

Node-API provides a set of APIs that allow JavaScript code to call back into native code. Node-APIs that support calling back into native code take in a callback functions represented by the `napi_callback` type. When the JavaScript VM calls back to native code, the `napi_callback` function provided is invoked. The APIs documented in this section allow the callback function to do the following:

- Get information about the context in which the callback was invoked.
- Get the arguments passed into the callback.
- Return a `napi_value` back from the callback.

Additionally, Node-API provides a set of functions which allow calling JavaScript functions from native code. One can either call a function like a regular JavaScript function call, or as a constructor function.

Any non-`NULL` data which is passed to this API via the `data` field of the `napi_property_descriptor` items can be associated with `object` and freed whenever `object` is garbage-collected by passing both `object` and the data to `napi_add_finalizer`.

`napi_call_function`

```
NAPI_EXTERN napi_status napi_call_function(napi_env env,
                                           napi_value recv,
                                           napi_value func,
                                           size_t argc,
                                           const napi_value* argv,
                                           napi_value* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] recv`: The `this` value passed to the called function.
- `[in] func`: `napi_value` representing the JavaScript function to be invoked.
- `[in] argc`: The count of elements in the `argv` array.
- `[in] argv`: Array of `napi_values` representing JavaScript values passed in as arguments to the function.
- `[out] result`: `napi_value` representing the JavaScript object returned.

Returns `napi_ok` if the API succeeded.

This method allows a JavaScript function object to be called from a native add-on. This is the primary mechanism of calling back *from* the add-on's native code *into* JavaScript. For the special case of calling into JavaScript after an async operation, see `napi_make_callback`.

A sample use case might look as follows. Consider the following JavaScript snippet:

```
function AddTwo(num) {
  return num + 2;
}
global.AddTwo = AddTwo;
```

Then, the above function can be invoked from a native add-on using the following code:

```
// Get the function named "AddTwo" on the global object
napi_value global, add_two, arg;
napi_status status = napi_get_global(env, &global);
if (status != napi_ok) return;

status = napi_get_named_property(env, global, "AddTwo", &add_two);
if (status != napi_ok) return;

// const arg = 1337
status = napi_create_int32(env, 1337, &arg);
if (status != napi_ok) return;

napi_value* argv = &arg;
size_t argc = 1;
```

```
// AddTwo(arg);
napi_value return_val;
status = napi_call_function(env, global, add_two, argc, argv, &return_val);
if (status != napi_ok) return;

// Convert the result back to a native type
int32_t result;
status = napi_get_value_int32(env, return_val, &result);
if (status != napi_ok) return;
```

napi_create_function

```
napi_status napi_create_function(napi_env env,
                                 const char* utf8name,
                                 size_t length,
                                 napi_callback cb,
                                 void* data,
                                 napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] utf8Name` : Optional name of the function encoded as UTF8. This is visible within JavaScript as the new function object's `name` property.
- `[in] length` : The length of the `utf8name` in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] cb` : The native function which should be called when this function object is invoked. `napi_callback` provides more details.
- `[in] data` : User-provided data context. This will be passed back into the function when invoked later.
- `[out] result` : `napi_value` representing the JavaScript function object for the newly created function.

Returns `napi_ok` if the API succeeded.

This API allows an add-on author to create a function object in native code. This is the primary mechanism to allow calling *into* the add-on's native code *from* JavaScript.

The newly created function is not automatically visible from script after this call. Instead, a property must be explicitly set on any object that is visible to JavaScript, in order for the function to be accessible from script.

In order to expose a function as part of the add-on's module exports, set the newly created function on the exports object. A sample module might look as follows:

```
napi_value SayHello(napi_env env, napi_callback_info info) {
  printf("Hello\n");
  return NULL;
}

napi_value Init(napi_env env, napi_value exports) {
  napi_status status;

  napi_value fn;
```

```
  status = napi_create_function(env, NULL, 0, SayHello, NULL, &fn);
  if (status != napi_ok) return NULL;

  status = napi_set_named_property(env, exports, "sayHello", fn);
  if (status != napi_ok) return NULL;

  return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)
```

Given the above code, the add-on can be used from JavaScript as follows:

```
const myaddon = require('./addon');
myaddon.sayHello();
```

The string passed to `require()` is the name of the target in `binding.gyp` responsible for creating the `.node` file.

Any non-`NULL` data which is passed to this API via the `data` parameter can be associated with the resulting JavaScript function (which is returned in the `result` parameter) and freed whenever the function is garbage-collected by passing both the JavaScript function and the data to `napi_add_finalizer`.

JavaScript `Function`s are described in Section 19.2 of the ECMAScript Language Specification.

`napi_get_cb_info`

```
napi_status napi_get_cb_info(napi_env env,
                             napi_callback_info cbinfo,
                             size_t* argc,
                             napi_value* argv,
                             napi_value* thisArg,
                             void** data)
```

- `[in] env`: The environment that the API is invoked under.
- `[in] cbinfo`: The callback info passed into the callback function.
- `[in-out] argc`: Specifies the length of the provided `argv` array and receives the actual count of arguments. `argc` can optionally be ignored by passing `NULL`.
- `[out] argv`: C array of `napi_value`s to which the arguments will be copied. If there are more arguments than the provided count, only the requested number of arguments are copied. If there are fewer arguments provided than claimed, the rest of `argv` is filled with `napi_value` values that represent `undefined`. `argv` can optionally be ignored by passing `NULL`.
- `[out] this`: Receives the JavaScript `this` argument for the call. `this` can optionally be ignored by passing `NULL`.
- `[out] data`: Receives the data pointer for the callback. `data` can optionally be ignored by passing `NULL`.

Returns `napi_ok` if the API succeeded.

This method is used within a callback function to retrieve details about the call like the arguments and the `this` pointer from a given callback info.

`napi_get_new_target`

```
napi_status napi_get_new_target(napi_env env,
                                napi_callback_info cbinfo,
                                napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] cbinfo` : The callback info passed into the callback function.
- `[out] result` : The `new.target` of the constructor call.

Returns `napi_ok` if the API succeeded.

This API returns the `new.target` of the constructor call. If the current callback is not a constructor call, the result is `NULL` .

`napi_new_instance`

```
napi_status napi_new_instance(napi_env env,
                              napi_value cons,
                              size_t argc,
                              napi_value* argv,
                              napi_value* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] cons` : `napi_value` representing the JavaScript function to be invoked as a constructor.
- `[in] argc` : The count of elements in the `argv` array.
- `[in] argv` : Array of JavaScript values as `napi_value` representing the arguments to the constructor. If `argc` is zero this parameter may be omitted by passing in `NULL` .
- `[out] result` : `napi_value` representing the JavaScript object returned, which in this case is the constructed object.

This method is used to instantiate a new JavaScript value using a given `napi_value` that represents the constructor for the object. For example, consider the following snippet:

```
function MyObject(param) {
  this.param = param;
}

const arg = 'hello';
const value = new MyObject(arg);
```

The following can be approximated in Node-API using the following snippet:

```
// Get the constructor function MyObject
napi_value global, constructor, arg, value;
napi_status status = napi_get_global(env, &global);
```

```
if (status != napi_ok) return;

status = napi_get_named_property(env, global, "MyObject", &constructor);
if (status != napi_ok) return;

// const arg = "hello"
status = napi_create_string_utf8(env, "hello", NAPI_AUTO_LENGTH, &arg);
if (status != napi_ok) return;

napi_value* argv = &arg;
size_t argc = 1;

// const value = new MyObject(arg)
status = napi_new_instance(env, constructor, argc, argv, &value);
```

Returns `napi_ok` if the API succeeded.

Object wrap

Node-API offers a way to "wrap" C++ classes and instances so that the class constructor and methods can be called from JavaScript.

1. The `napi_define_class` API defines a JavaScript class with constructor, static properties and methods, and instance properties and methods that correspond to the C++ class.
2. When JavaScript code invokes the constructor, the constructor callback uses `napi_wrap` to wrap a new C++ instance in a JavaScript object, then returns the wrapper object.
3. When JavaScript code invokes a method or property accessor on the class, the corresponding `napi_callback` C++ function is invoked. For an instance callback, `napi_unwrap` obtains the C++ instance that is the target of the call.

For wrapped objects it may be difficult to distinguish between a function called on a class prototype and a function called on an instance of a class. A common pattern used to address this problem is to save a persistent reference to the class constructor for later `instanceof` checks.

```
napi_value MyClass_constructor = NULL;
status = napi_get_reference_value(env, MyClass::es_constructor,
&MyClass_constructor);
assert(napi_ok == status);
bool is_instance = false;
status = napi_instanceof(env, es_this, MyClass_constructor, &is_instance);
assert(napi_ok == status);
if (is_instance) {
  // napi_unwrap() ...
} else {
  // otherwise...
}
```

The reference must be freed once it is no longer needed.

There are occasions where `napi_instanceof()` is insufficient for ensuring that a JavaScript object is a wrapper for a certain native type. This is the case especially when wrapped JavaScript objects are passed back into the addon

via static methods rather than as the `this` value of prototype methods. In such cases there is a chance that they may be unwrapped incorrectly.

```
const myAddon = require('./build/Release/my_addon.node');

// `openDatabase()` returns a JavaScript object that wraps a native database
// handle.
const dbHandle = myAddon.openDatabase();

// `query()` returns a JavaScript object that wraps a native query handle.
const queryHandle = myAddon.query(dbHandle, 'Gimme ALL the things!');

// There is an accidental error in the line below. The first parameter to
// `myAddon.queryHasRecords()` should be the database handle (`dbHandle`), not
// the query handle (`query`), so the correct condition for the while-loop
// should be
//
// myAddon.queryHasRecords(dbHandle, queryHandle)
//
while (myAddon.queryHasRecords(queryHandle, dbHandle)) {
  // retrieve records
}
```

In the above example `myAddon.queryHasRecords()` is a method that accepts two arguments. The first is a database handle and the second is a query handle. Internally, it unwraps the first argument and casts the resulting pointer to a native database handle. It then unwraps the second argument and casts the resulting pointer to a query handle. If the arguments are passed in the wrong order, the casts will work, however, there is a good chance that the underlying database operation will fail, or will even cause an invalid memory access.

To ensure that the pointer retrieved from the first argument is indeed a pointer to a database handle and, similarly, that the pointer retrieved from the second argument is indeed a pointer to a query handle, the implementation of `queryHasRecords()` has to perform a type validation. Retaining the JavaScript class constructor from which the database handle was instantiated and the constructor from which the query handle was instantiated in `napi_ref`s can help, because `napi_instanceof()` can then be used to ensure that the instances passed into `queryHashRecords()` are indeed of the correct type.

Unfortunately, `napi_instanceof()` does not protect against prototype manipulation. For example, the prototype of the database handle instance can be set to the prototype of the constructor for query handle instances. In this case, the database handle instance can appear as a query handle instance, and it will pass the `napi_instanceof()` test for a query handle instance, while still containing a pointer to a database handle.

To this end, Node-API provides type-tagging capabilities.

A type tag is a 128-bit integer unique to the addon. Node-API provides the `napi_type_tag` structure for storing a type tag. When such a value is passed along with a JavaScript object stored in a `napi_value` to `napi_type_tag_object()`, the JavaScript object will be "marked" with the type tag. The "mark" is invisible on the JavaScript side. When a JavaScript object arrives into a native binding, `napi_check_object_type_tag()` can be used along with the original type tag to determine whether the JavaScript object was previously "marked" with the type tag. This creates a type-checking capability of a higher fidelity than `napi_instanceof()` can provide, because such type- tagging survives prototype manipulation and addon unloading/reloading.

Continuing the above example, the following skeleton addon implementation illustrates the use of `napi_type_tag_object()` and `napi_check_object_type_tag()`.

```
// This value is the type tag for a database handle. The command
//
//   uuidgen | sed -r -e 's/-//g' -e 's/(.{16})(.*)/0x\1, 0x\2/'
//
// can be used to obtain the two values with which to initialize the structure.
static const napi_type_tag DatabaseHandleTypeTag = {
  0x1edf75a38336451d, 0xa5ed9ce2e4c00c38
};

// This value is the type tag for a query handle.
static const napi_type_tag QueryHandleTypeTag = {
  0x9c73317f9fad44a3, 0x93c3920bf3b0ad6a
};

static napi_value
openDatabase(napi_env env, napi_callback_info info) {
  napi_status status;
  napi_value result;

  // Perform the underlying action which results in a database handle.
  DatabaseHandle* dbHandle = open_database();

  // Create a new, empty JS object.
  status = napi_create_object(env, &result);
  if (status != napi_ok) return NULL;

  // Tag the object to indicate that it holds a pointer to a `DatabaseHandle`.
  status = napi_type_tag_object(env, result, &DatabaseHandleTypeTag);
  if (status != napi_ok) return NULL;

  // Store the pointer to the `DatabaseHandle` structure inside the JS object.
  status = napi_wrap(env, result, dbHandle, NULL, NULL, NULL);
  if (status != napi_ok) return NULL;

  return result;
}

// Later when we receive a JavaScript object purporting to be a database handle
// we can use `napi_check_object_type_tag()` to ensure that it is indeed such a
// handle.

static napi_value
query(napi_env env, napi_callback_info info) {
  napi_status status;
  size_t argc = 2;
  napi_value argv[2];
  bool is_db_handle;
```

```
  status = napi_get_cb_info(env, info, &argc, argv, NULL, NULL);
  if (status != napi_ok) return NULL;

  // Check that the object passed as the first parameter has the previously
  // applied tag.
  status = napi_check_object_type_tag(env,
                                      argv[0],
                                      &DatabaseHandleTypeTag,
                                      &is_db_handle);
  if (status != napi_ok) return NULL;

  // Throw a `TypeError` if it doesn't.
  if (!is_db_handle) {
    // Throw a TypeError.
    return NULL;
  }
}
```

`napi_define_class`

```
napi_status napi_define_class(napi_env env,
                              const char* utf8name,
                              size_t length,
                              napi_callback constructor,
                              void* data,
                              size_t property_count,
                              const napi_property_descriptor* properties,
                              napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] utf8name` : Name of the JavaScript constructor function; When wrapping a C++ class, we recommend for clarity that this name be the same as that of the C++ class.
- `[in] length` : The length of the `utf8name` in bytes, or `NAPI_AUTO_LENGTH` if it is null-terminated.
- `[in] constructor` : Callback function that handles constructing instances of the class. When wrapping a C++ class, this method must be a static member with the `napi_callback` signature. A C++ class constructor cannot be used. `napi_callback` provides more details.
- `[in] data` : Optional data to be passed to the constructor callback as the `data` property of the callback info.
- `[in] property_count` : Number of items in the `properties` array argument.
- `[in] properties` : Array of property descriptors describing static and instance data properties, accessors, and methods on the class See `napi_property_descriptor` .
- `[out] result` : A `napi_value` representing the constructor function for the class.

Returns `napi_ok` if the API succeeded.

Defines a JavaScript class, including:

- A JavaScript constructor function that has the class name. When wrapping a corresponding C++ class, the callback passed via `constructor` can be used to instantiate a new C++ class instance, which can then be placed inside the JavaScript object instance being constructed using `napi_wrap` .

- Properties on the constructor function whose implementation can call corresponding *static* data properties, accessors, and methods of the C++ class (defined by property descriptors with the `napi_static` attribute).
- Properties on the constructor function's `prototype` object. When wrapping a C++ class, *non-static* data properties, accessors, and methods of the C++ class can be called from the static functions given in the property descriptors without the `napi_static` attribute after retrieving the C++ class instance placed inside the JavaScript object instance by using `napi_unwrap`.

When wrapping a C++ class, the C++ constructor callback passed via `constructor` should be a static method on the class that calls the actual class constructor, then wraps the new C++ instance in a JavaScript object, and returns the wrapper object. See `napi_wrap` for details.

The JavaScript constructor function returned from `napi_define_class` is often saved and used later to construct new instances of the class from native code, and/or to check whether provided values are instances of the class. In that case, to prevent the function value from being garbage-collected, a strong persistent reference to it can be created using `napi_create_reference`, ensuring that the reference count is kept >= 1.

Any non-`NULL` data which is passed to this API via the `data` parameter or via the `data` field of the `napi_property_descriptor` array items can be associated with the resulting JavaScript constructor (which is returned in the `result` parameter) and freed whenever the class is garbage-collected by passing both the JavaScript function and the data to `napi_add_finalizer`.

`napi_wrap`

```
napi_status napi_wrap(napi_env env,
                      napi_value js_object,
                      void* native_object,
                      napi_finalize finalize_cb,
                      void* finalize_hint,
                      napi_ref* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] js_object`: The JavaScript object that will be the wrapper for the native object.
- `[in] native_object`: The native instance that will be wrapped in the JavaScript object.
- `[in] finalize_cb`: Optional native callback that can be used to free the native instance when the JavaScript object is ready for garbage-collection. `napi_finalize` provides more details.
- `[in] finalize_hint`: Optional contextual hint that is passed to the finalize callback.
- `[out] result`: Optional reference to the wrapped object.

Returns `napi_ok` if the API succeeded.

Wraps a native instance in a JavaScript object. The native instance can be retrieved later using `napi_unwrap()`.

When JavaScript code invokes a constructor for a class that was defined using `napi_define_class()`, the `napi_callback` for the constructor is invoked. After constructing an instance of the native class, the callback must then call `napi_wrap()` to wrap the newly constructed instance in the already-created JavaScript object that is the `this` argument to the constructor callback. (That `this` object was created from the constructor function's `prototype`, so it already has definitions of all the instance properties and methods.)

Typically when wrapping a class instance, a finalize callback should be provided that simply deletes the native instance that is received as the `data` argument to the finalize callback.

The optional returned reference is initially a weak reference, meaning it has a reference count of 0. Typically this reference count would be incremented temporarily during async operations that require the instance to remain valid.

Caution: The optional returned reference (if obtained) should be deleted via `napi_delete_reference` ONLY in response to the finalize callback invocation. If it is deleted before then, then the finalize callback may never be invoked. Therefore, when obtaining a reference a finalize callback is also required in order to enable correct disposal of the reference.

Finalizer callbacks may be deferred, leaving a window where the object has been garbage collected (and the weak reference is invalid) but the finalizer hasn't been called yet. When using `napi_get_reference_value()` on weak references returned by `napi_wrap()`, you should still handle an empty result.

Calling `napi_wrap()` a second time on an object will return an error. To associate another native instance with the object, use `napi_remove_wrap()` first.

`napi_unwrap`

```
napi_status napi_unwrap(napi_env env,
                        napi_value js_object,
                        void** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] js_object`: The object associated with the native instance.
- `[out] result`: Pointer to the wrapped native instance.

Returns `napi_ok` if the API succeeded.

Retrieves a native instance that was previously wrapped in a JavaScript object using `napi_wrap()`.

When JavaScript code invokes a method or property accessor on the class, the corresponding `napi_callback` is invoked. If the callback is for an instance method or accessor, then the `this` argument to the callback is the wrapper object; the wrapped C++ instance that is the target of the call can be obtained then by calling `napi_unwrap()` on the wrapper object.

`napi_remove_wrap`

```
napi_status napi_remove_wrap(napi_env env,
                             napi_value js_object,
                             void** result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] js_object`: The object associated with the native instance.
- `[out] result`: Pointer to the wrapped native instance.

Returns `napi_ok` if the API succeeded.

Retrieves a native instance that was previously wrapped in the JavaScript object `js_object` using `napi_wrap()` and removes the wrapping. If a finalize callback was associated with the wrapping, it will no longer be called when

the JavaScript object becomes garbage-collected.

`napi_type_tag_object`

```
napi_status napi_type_tag_object(napi_env env,
                                 napi_value js_object,
                                 const napi_type_tag* type_tag);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The JavaScript object to be marked.
- `[in] type_tag` : The tag with which the object is to be marked.

Returns `napi_ok` if the API succeeded.

Associates the value of the `type_tag` pointer with the JavaScript object. `napi_check_object_type_tag()` can then be used to compare the tag that was attached to the object with one owned by the addon to ensure that the object has the right type.

If the object already has an associated type tag, this API will return `napi_invalid_arg` .

`napi_check_object_type_tag`

```
napi_status napi_check_object_type_tag(napi_env env,
                                       napi_value js_object,
                                       const napi_type_tag* type_tag,
                                       bool* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] js_object` : The JavaScript object whose type tag to examine.
- `[in] type_tag` : The tag with which to compare any tag found on the object.
- `[out] result` : Whether the type tag given matched the type tag on the object. `false` is also returned if no type tag was found on the object.

Returns `napi_ok` if the API succeeded.

Compares the pointer given as `type_tag` with any that can be found on `js_object` . If no tag is found on `js_object` or, if a tag is found but it does not match `type_tag` , then `result` is set to `false` . If a tag is found and it matches `type_tag` , then `result` is set to `true` .

`napi_add_finalizer`

```
napi_status napi_add_finalizer(napi_env env,
                               napi_value js_object,
                               void* native_object,
                               napi_finalize finalize_cb,
                               void* finalize_hint,
                               napi_ref* result);
```

- `[in] env` : The environment that the API is invoked under.

- `[in] js_object` : The JavaScript object to which the native data will be attached.
- `[in] native_object` : The native data that will be attached to the JavaScript object.
- `[in] finalize_cb` : Native callback that will be used to free the native data when the JavaScript object is ready for garbage-collection. `napi_finalize` provides more details.
- `[in] finalize_hint` : Optional contextual hint that is passed to the finalize callback.
- `[out] result` : Optional reference to the JavaScript object.

Returns `napi_ok` if the API succeeded.

Adds a `napi_finalize` callback which will be called when the JavaScript object in `js_object` is ready for garbage collection. This API is similar to `napi_wrap()` except that:

- the native data cannot be retrieved later using `napi_unwrap()` ,
- nor can it be removed later using `napi_remove_wrap()` , and
- the API can be called multiple times with different data items in order to attach each of them to the JavaScript object, and
- the object manipulated by the API can be used with `napi_wrap()` .

Caution: The optional returned reference (if obtained) should be deleted via `napi_delete_reference` ONLY in response to the finalize callback invocation. If it is deleted before then, then the finalize callback may never be invoked. Therefore, when obtaining a reference a finalize callback is also required in order to enable correct disposal of the reference.

Simple asynchronous operations

Addon modules often need to leverage async helpers from libuv as part of their implementation. This allows them to schedule work to be executed asynchronously so that their methods can return in advance of the work being completed. This allows them to avoid blocking overall execution of the Node.js application.

Node-API provides an ABI-stable interface for these supporting functions which covers the most common asynchronous use cases.

Node-API defines the `napi_async_work` structure which is used to manage asynchronous workers. Instances are created/deleted with `napi_create_async_work` and `napi_delete_async_work` .

The `execute` and `complete` callbacks are functions that will be invoked when the executor is ready to execute and when it completes its task respectively.

The `execute` function should avoid making any Node-API calls that could result in the execution of JavaScript or interaction with JavaScript objects. Most often, any code that needs to make Node-API calls should be made in `complete` callback instead. Avoid using the `napi_env` parameter in the execute callback as it will likely execute JavaScript.

These functions implement the following interfaces:

```
typedef void (*napi_async_execute_callback)(napi_env env,
                                            void* data);
typedef void (*napi_async_complete_callback)(napi_env env,
                                             napi_status status,
                                             void* data);
```

When these methods are invoked, the `data` parameter passed will be the addon-provided `void*` data that was passed into the `napi_create_async_work` call.

Once created the async worker can be queued for execution using the `napi_queue_async_work` function:

```
napi_status napi_queue_async_work(napi_env env,
                                  napi_async_work work);
```

`napi_cancel_async_work` can be used if the work needs to be cancelled before the work has started execution.

After calling `napi_cancel_async_work`, the `complete` callback will be invoked with a status value of `napi_cancelled`. The work should not be deleted before the `complete` callback invocation, even when it was cancelled.

napi_create_async_work

```
napi_status napi_create_async_work(napi_env env,
                                   napi_value async_resource,
                                   napi_value async_resource_name,
                                   napi_async_execute_callback execute,
                                   napi_async_complete_callback complete,
                                   void* data,
                                   napi_async_work* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] async_resource`: An optional object associated with the async work that will be passed to possible `async_hooks` `init` hooks.
- `[in] async_resource_name`: Identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- `[in] execute`: The native function which should be called to execute the logic asynchronously. The given function is called from a worker pool thread and can execute in parallel with the main event loop thread.
- `[in] complete`: The native function which will be called when the asynchronous logic is completed or is cancelled. The given function is called from the main event loop thread. `napi_async_complete_callback` provides more details.
- `[in] data`: User-provided data context. This will be passed back into the execute and complete functions.
- `[out] result`: `napi_async_work*` which is the handle to the newly created async work.

Returns `napi_ok` if the API succeeded.

This API allocates a work object that is used to execute logic asynchronously. It should be freed using `napi_delete_async_work` once the work is no longer required.

`async_resource_name` should be a null-terminated, UTF-8-encoded string.

The `async_resource_name` identifier is provided by the user and should be representative of the type of async work being performed. It is also recommended to apply namespacing to the identifier, e.g. by including the module name. See the `async_hooks` documentation for more information.

`napi_delete_async_work`

```
napi_status napi_delete_async_work(napi_env env,
                                   napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work` .

Returns `napi_ok` if the API succeeded.

This API frees a previously allocated work object.

This API can be called even if there is a pending JavaScript exception.

`napi_queue_async_work`

```
napi_status napi_queue_async_work(napi_env env,
                                  napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work` .

Returns `napi_ok` if the API succeeded.

This API requests that the previously allocated work be scheduled for execution. Once it returns successfully, this API must not be called again with the same `napi_async_work` item or the result will be undefined.

`napi_cancel_async_work`

```
napi_status napi_cancel_async_work(napi_env env,
                                   napi_async_work work);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] work` : The handle returned by the call to `napi_create_async_work` .

Returns `napi_ok` if the API succeeded.

This API cancels queued work if it has not yet been started. If it has already started executing, it cannot be cancelled and `napi_generic_failure` will be returned. If successful, the `complete` callback will be invoked with a status value of `napi_cancelled` . The work should not be deleted before the `complete` callback invocation, even if it has been successfully cancelled.

This API can be called even if there is a pending JavaScript exception.

Custom asynchronous operations

The simple asynchronous work APIs above may not be appropriate for every scenario. When using any other asynchronous mechanism, the following APIs are necessary to ensure an asynchronous operation is properly tracked by the runtime.

`napi_async_init`

```
napi_status napi_async_init(napi_env env,
                            napi_value async_resource,
                            napi_value async_resource_name,
                            napi_async_context* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_resource` : Object associated with the async work that will be passed to possible `async_hooks` `init` hooks and can be accessed by `async_hooks.executionAsyncResource()` .
- `[in] async_resource_name` : Identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- `[out] result` : The initialized async context.

Returns `napi_ok` if the API succeeded.

The `async_resource` object needs to be kept alive until `napi_async_destroy` to keep `async_hooks` related API acts correctly. In order to retain ABI compatibility with previous versions, `napi_async_context` s are not maintaining the strong reference to the `async_resource` objects to avoid introducing causing memory leaks. However, if the `async_resource` is garbage collected by JavaScript engine before the `napi_async_context` was destroyed by `napi_async_destroy` , calling `napi_async_context` related APIs like `napi_open_callback_scope` and `napi_make_callback` can cause problems like loss of async context when using the `AsyncLocalStorage` API.

In order to retain ABI compatibility with previous versions, passing `NULL` for `async_resource` does not result in an error. However, this is not recommended as this will result poor results with `async_hooks` `init` hooks and `async_hooks.executionAsyncResource()` as the resource is now required by the underlying `async_hooks` implementation in order to provide the linkage between async callbacks.

`napi_async_destroy`

```
napi_status napi_async_destroy(napi_env env,
                               napi_async_context async_context);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_context` : The async context to be destroyed.

Returns `napi_ok` if the API succeeded.

This API can be called even if there is a pending JavaScript exception.

`napi_make_callback`

```
NAPI_EXTERN napi_status napi_make_callback(napi_env env,
                                           napi_async_context async_context,
                                           napi_value recv,
                                           napi_value func,
                                           size_t argc,
```

```
                                          const napi_value* argv,
                                          napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] async_context` : Context for the async operation that is invoking the callback. This should normally be a value previously obtained from `napi_async_init` . In order to retain ABI compatibility with previous versions, passing `NULL` for `async_context` does not result in an error. However, this results in incorrect operation of async hooks. Potential issues include loss of async context when using the `AsyncLocalStorage` API.
- `[in] recv` : The `this` value passed to the called function.
- `[in] func` : `napi_value` representing the JavaScript function to be invoked.
- `[in] argc` : The count of elements in the `argv` array.
- `[in] argv` : Array of JavaScript values as `napi_value` representing the arguments to the function. If `argc` is zero this parameter may be omitted by passing in `NULL` .
- `[out] result` : `napi_value` representing the JavaScript object returned.

Returns `napi_ok` if the API succeeded.

This method allows a JavaScript function object to be called from a native add-on. This API is similar to `napi_call_function` . However, it is used to call *from* native code back *into* JavaScript *after* returning from an async operation (when there is no other script on the stack). It is a fairly simple wrapper around `node::MakeCallback` .

Note it is *not* necessary to use `napi_make_callback` from within a `napi_async_complete_callback` ; in that situation the callback's async context has already been set up, so a direct call to `napi_call_function` is sufficient and appropriate. Use of the `napi_make_callback` function may be required when implementing custom async behavior that does not use `napi_create_async_work` .

Any `process.nextTick` s or Promises scheduled on the microtask queue by JavaScript during the callback are ran before returning back to C/C++.

`napi_open_callback_scope`

```
NAPI_EXTERN napi_status napi_open_callback_scope(napi_env env,
                                                 napi_value resource_object,
                                                 napi_async_context context,
                                                 napi_callback_scope* result)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] resource_object` : An object associated with the async work that will be passed to possible `async_hooks` `init` hooks. This parameter has been deprecated and is ignored at runtime. Use the `async_resource` parameter in `napi_async_init` instead.
- `[in] context` : Context for the async operation that is invoking the callback. This should be a value previously obtained from `napi_async_init` .
- `[out] result` : The newly created scope.

There are cases (for example, resolving promises) where it is necessary to have the equivalent of the scope associated with a callback in place when making certain Node-API calls. If there is no other script on the stack the

`napi_open_callback_scope` and `napi_close_callback_scope` functions can be used to open/close the required scope.

`napi_close_callback_scope`

```
NAPI_EXTERN napi_status napi_close_callback_scope(napi_env env,
                                                  napi_callback_scope scope)
```

- `[in] env` : The environment that the API is invoked under.
- `[in] scope` : The scope to be closed.

This API can be called even if there is a pending JavaScript exception.

Version management

`napi_get_node_version`

```
typedef struct {
  uint32_t major;
  uint32_t minor;
  uint32_t patch;
  const char* release;
} napi_node_version;

napi_status napi_get_node_version(napi_env env,
                                  const napi_node_version** version);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] version` : A pointer to version information for Node.js itself.

Returns `napi_ok` if the API succeeded.

This function fills the `version` struct with the major, minor, and patch version of Node.js that is currently running, and the `release` field with the value of `process.release.name` .

The returned buffer is statically allocated and does not need to be freed.

`napi_get_version`

```
napi_status napi_get_version(napi_env env,
                             uint32_t* result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : The highest version of Node-API supported.

Returns `napi_ok` if the API succeeded.

This API returns the highest Node-API version supported by the Node.js runtime. Node-API is planned to be additive such that newer releases of Node.js may support additional API functions. In order to allow an addon to use a newer

function when running with versions of Node.js that support it, while providing fallback behavior when running with Node.js versions that don't support it:

- Call `napi_get_version()` to determine if the API is available.
- If available, dynamically load a pointer to the function using `uv_dlsym()`.
- Use the dynamically loaded pointer to invoke the function.
- If the function is not available, provide an alternate implementation that does not use the function.

Memory management

`napi_adjust_external_memory`

```
NAPI_EXTERN napi_status napi_adjust_external_memory(napi_env env,
                                                    int64_t change_in_bytes,
                                                    int64_t* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] change_in_bytes`: The change in externally allocated memory that is kept alive by JavaScript objects.
- `[out] result`: The adjusted value

Returns `napi_ok` if the API succeeded.

This function gives V8 an indication of the amount of externally allocated memory that is kept alive by JavaScript objects (i.e. a JavaScript object that points to its own memory allocated by a native module). Registering externally allocated memory will trigger global garbage collections more often than it would otherwise.

Promises

Node-API provides facilities for creating `Promise` objects as described in Section 25.4 of the ECMA specification. It implements promises as a pair of objects. When a promise is created by `napi_create_promise()`, a "deferred" object is created and returned alongside the `Promise`. The deferred object is bound to the created `Promise` and is the only means to resolve or reject the `Promise` using `napi_resolve_deferred()` or `napi_reject_deferred()`. The deferred object that is created by `napi_create_promise()` is freed by `napi_resolve_deferred()` or `napi_reject_deferred()`. The `Promise` object may be returned to JavaScript where it can be used in the usual fashion.

For example, to create a promise and pass it to an asynchronous worker:

```
napi_deferred deferred;
napi_value promise;
napi_status status;

// Create the promise.
status = napi_create_promise(env, &deferred, &promise);
if (status != napi_ok) return NULL;

// Pass the deferred to a function that performs an asynchronous action.
do_something_asynchronous(deferred);
```

```
// Return the promise to JS
return promise;
```

The above function `do_something_asynchronous()` would perform its asynchronous action and then it would resolve or reject the deferred, thereby concluding the promise and freeing the deferred:

```
napi_deferred deferred;
napi_value undefined;
napi_status status;

// Create a value with which to conclude the deferred.
status = napi_get_undefined(env, &undefined);
if (status != napi_ok) return NULL;

// Resolve or reject the promise associated with the deferred depending on
// whether the asynchronous action succeeded.
if (asynchronous_action_succeeded) {
  status = napi_resolve_deferred(env, deferred, undefined);
} else {
  status = napi_reject_deferred(env, deferred, undefined);
}
if (status != napi_ok) return NULL;

// At this point the deferred has been freed, so we should assign NULL to it.
deferred = NULL;
```

`napi_create_promise`

```
napi_status napi_create_promise(napi_env env,
                                napi_deferred* deferred,
                                napi_value* promise);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] deferred` : A newly created deferred object which can later be passed to `napi_resolve_deferred()` or `napi_reject_deferred()` to resolve resp. reject the associated promise.
- `[out] promise` : The JavaScript promise associated with the deferred object.

Returns `napi_ok` if the API succeeded.

This API creates a deferred object and a JavaScript promise.

`napi_resolve_deferred`

```
napi_status napi_resolve_deferred(napi_env env,
                                  napi_deferred deferred,
                                  napi_value resolution);
```

- `[in] env` : The environment that the API is invoked under.

- `[in] deferred` : The deferred object whose associated promise to resolve.
- `[in] resolution` : The value with which to resolve the promise.

This API resolves a JavaScript promise by way of the deferred object with which it is associated. Thus, it can only be used to resolve JavaScript promises for which the corresponding deferred object is available. This effectively means that the promise must have been created using `napi_create_promise()` and the deferred object returned from that call must have been retained in order to be passed to this API.

The deferred object is freed upon successful completion.

`napi_reject_deferred`

```
napi_status napi_reject_deferred(napi_env env,
                                 napi_deferred deferred,
                                 napi_value rejection);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] deferred` : The deferred object whose associated promise to resolve.
- `[in] rejection` : The value with which to reject the promise.

This API rejects a JavaScript promise by way of the deferred object with which it is associated. Thus, it can only be used to reject JavaScript promises for which the corresponding deferred object is available. This effectively means that the promise must have been created using `napi_create_promise()` and the deferred object returned from that call must have been retained in order to be passed to this API.

The deferred object is freed upon successful completion.

`napi_is_promise`

```
napi_status napi_is_promise(napi_env env,
                            napi_value value,
                            bool* is_promise);
```

- `[in] env` : The environment that the API is invoked under.
- `[in] value` : The value to examine
- `[out] is_promise` : Flag indicating whether `promise` is a native promise object (that is, a promise object created by the underlying engine).

Script execution

Node-API provides an API for executing a string containing JavaScript using the underlying JavaScript engine.

`napi_run_script`

```
NAPI_EXTERN napi_status napi_run_script(napi_env env,
                                        napi_value script,
                                        napi_value* result);
```

- `[in] env` : The environment that the API is invoked under.

- `[in] script` : A JavaScript string containing the script to execute.
- `[out] result` : The value resulting from having executed the script.

This function executes a string of JavaScript code and returns its result with the following caveats:

- Unlike `eval` , this function does not allow the script to access the current lexical scope, and therefore also does not allow to access the module scope, meaning that pseudo-globals such as `require` will not be available.
- The script can access the global scope. Function and `var` declarations in the script will be added to the `global` object. Variable declarations made using `let` and `const` will be visible globally, but will not be added to the `global` object.
- The value of `this` is `global` within the script.

libuv event loop

Node-API provides a function for getting the current event loop associated with a specific `napi_env` .

`napi_get_uv_event_loop`

```
NAPI_EXTERN napi_status napi_get_uv_event_loop(napi_env env,
                                               struct uv_loop_s** loop);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] loop` : The current libuv loop instance.

Asynchronous thread-safe function calls

JavaScript functions can normally only be called from a native addon's main thread. If an addon creates additional threads, then Node-API functions that require a `napi_env` , `napi_value` , or `napi_ref` must not be called from those threads.

When an addon has additional threads and JavaScript functions need to be invoked based on the processing completed by those threads, those threads must communicate with the addon's main thread so that the main thread can invoke the JavaScript function on their behalf. The thread-safe function APIs provide an easy way to do this.

These APIs provide the type `napi_threadsafe_function` as well as APIs to create, destroy, and call objects of this type. `napi_create_threadsafe_function()` creates a persistent reference to a `napi_value` that holds a JavaScript function which can be called from multiple threads. The calls happen asynchronously. This means that values with which the JavaScript callback is to be called will be placed in a queue, and, for each value in the queue, a call will eventually be made to the JavaScript function.

Upon creation of a `napi_threadsafe_function` a `napi_finalize` callback can be provided. This callback will be invoked on the main thread when the thread-safe function is about to be destroyed. It receives the context and the finalize data given during construction, and provides an opportunity for cleaning up after the threads e.g. by calling `uv_thread_join()` . **Aside from the main loop thread, no threads should be using the thread-safe function after the finalize callback completes.**

The `context` given during the call to `napi_create_threadsafe_function()` can be retrieved from any thread with a call to `napi_get_threadsafe_function_context()` .

Calling a thread-safe function

`napi_call_threadsafe_function()` can be used for initiating a call into JavaScript. `napi_call_threadsafe_function()` accepts a parameter which controls whether the API behaves blockingly. If set to `napi_tsfn_nonblocking`, the API behaves non-blockingly, returning `napi_queue_full` if the queue was full, preventing data from being successfully added to the queue. If set to `napi_tsfn_blocking`, the API blocks until space becomes available in the queue. `napi_call_threadsafe_function()` never blocks if the thread-safe function was created with a maximum queue size of 0.

`napi_call_threadsafe_function()` should not be called with `napi_tsfn_blocking` from a JavaScript thread, because, if the queue is full, it may cause the JavaScript thread to deadlock.

The actual call into JavaScript is controlled by the callback given via the `call_js_cb` parameter. `call_js_cb` is invoked on the main thread once for each value that was placed into the queue by a successful call to `napi_call_threadsafe_function()`. If such a callback is not given, a default callback will be used, and the resulting JavaScript call will have no arguments. The `call_js_cb` callback receives the JavaScript function to call as a `napi_value` in its parameters, as well as the `void*` context pointer used when creating the `napi_threadsafe_function`, and the next data pointer that was created by one of the secondary threads. The callback can then use an API such as `napi_call_function()` to call into JavaScript.

The callback may also be invoked with `env` and `call_js_cb` both set to `NULL` to indicate that calls into JavaScript are no longer possible, while items remain in the queue that may need to be freed. This normally occurs when the Node.js process exits while there is a thread-safe function still active.

It is not necessary to call into JavaScript via `napi_make_callback()` because Node-API runs `call_js_cb` in a context appropriate for callbacks.

Reference counting of thread-safe functions

Threads can be added to and removed from a `napi_threadsafe_function` object during its existence. Thus, in addition to specifying an initial number of threads upon creation, `napi_acquire_threadsafe_function` can be called to indicate that a new thread will start making use of the thread-safe function. Similarly, `napi_release_threadsafe_function` can be called to indicate that an existing thread will stop making use of the thread-safe function.

`napi_threadsafe_function` objects are destroyed when every thread which uses the object has called `napi_release_threadsafe_function()` or has received a return status of `napi_closing` in response to a call to `napi_call_threadsafe_function`. The queue is emptied before the `napi_threadsafe_function` is destroyed. `napi_release_threadsafe_function()` should be the last API call made in conjunction with a given `napi_threadsafe_function`, because after the call completes, there is no guarantee that the `napi_threadsafe_function` is still allocated. For the same reason, do not use a thread-safe function after receiving a return value of `napi_closing` in response to a call to `napi_call_threadsafe_function`. Data associated with the `napi_threadsafe_function` can be freed in its `napi_finalize` callback which was passed to `napi_create_threadsafe_function()`. The parameter `initial_thread_count` of `napi_create_threadsafe_function` marks the initial number of acquisitions of the thread-safe functions, instead of calling `napi_acquire_threadsafe_function` multiple times at creation.

Once the number of threads making use of a `napi_threadsafe_function` reaches zero, no further threads can start making use of it by calling `napi_acquire_threadsafe_function()`. In fact, all subsequent API calls associated with it, except `napi_release_threadsafe_function()`, will return an error value of `napi_closing`.

The thread-safe function can be "aborted" by giving a value of `napi_tsfn_abort` to `napi_release_threadsafe_function()`. This will cause all subsequent APIs associated with the thread-safe function except `napi_release_threadsafe_function()` to return `napi_closing` even before its reference count reaches zero. In particular, `napi_call_threadsafe_function()` will return `napi_closing`, thus informing the threads that it is no longer possible to make asynchronous calls to the thread-safe function. This can be used as a criterion for terminating the thread. **Upon receiving a return value of `napi_closing` from `napi_call_threadsafe_function()` a thread must not use the thread-safe function anymore because it is no longer guaranteed to be allocated.**

Deciding whether to keep the process running

Similarly to libuv handles, thread-safe functions can be "referenced" and "unreferenced". A "referenced" thread-safe function will cause the event loop on the thread on which it is created to remain alive until the thread-safe function is destroyed. In contrast, an "unreferenced" thread-safe function will not prevent the event loop from exiting. The APIs `napi_ref_threadsafe_function` and `napi_unref_threadsafe_function` exist for this purpose.

Neither does `napi_unref_threadsafe_function` mark the thread-safe functions as able to be destroyed nor does `napi_ref_threadsafe_function` prevent it from being destroyed.

`napi_create_threadsafe_function`

```
NAPI_EXTERN napi_status
napi_create_threadsafe_function(napi_env env,
                                napi_value func,
                                napi_value async_resource,
                                napi_value async_resource_name,
                                size_t max_queue_size,
                                size_t initial_thread_count,
                                void* thread_finalize_data,
                                napi_finalize thread_finalize_cb,
                                void* context,
                                napi_threadsafe_function_call_js call_js_cb,
                                napi_threadsafe_function* result);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] func`: An optional JavaScript function to call from another thread. It must be provided if `NULL` is passed to `call_js_cb`.
- `[in] async_resource`: An optional object associated with the async work that will be passed to possible `async_hooks` `init` hooks.
- `[in] async_resource_name`: A JavaScript string to provide an identifier for the kind of resource that is being provided for diagnostic information exposed by the `async_hooks` API.
- `[in] max_queue_size`: Maximum size of the queue. `0` for no limit.
- `[in] initial_thread_count`: The initial number of acquisitions, i.e. the initial number of threads, including the main thread, which will be making use of this function.
- `[in] thread_finalize_data`: Optional data to be passed to `thread_finalize_cb`.
- `[in] thread_finalize_cb`: Optional function to call when the `napi_threadsafe_function` is being destroyed.
- `[in] context`: Optional data to attach to the resulting `napi_threadsafe_function`.

- `[in] call_js_cb` : Optional callback which calls the JavaScript function in response to a call on a different thread. This callback will be called on the main thread. If not given, the JavaScript function will be called with no parameters and with `undefined` as its `this` value. `napi_threadsafe_function_call_js` provides more details.
- `[out] result` : The asynchronous thread-safe JavaScript function.

`napi_get_threadsafe_function_context`

```
NAPI_EXTERN napi_status
napi_get_threadsafe_function_context(napi_threadsafe_function func,
                                     void** result);
```

- `[in] func` : The thread-safe function for which to retrieve the context.
- `[out] result` : The location where to store the context.

This API may be called from any thread which makes use of `func` .

`napi_call_threadsafe_function`

```
NAPI_EXTERN napi_status
napi_call_threadsafe_function(napi_threadsafe_function func,
                              void* data,
                              napi_threadsafe_function_call_mode is_blocking);
```

- `[in] func` : The asynchronous thread-safe JavaScript function to invoke.
- `[in] data` : Data to send into JavaScript via the callback `call_js_cb` provided during the creation of the thread-safe JavaScript function.
- `[in] is_blocking` : Flag whose value can be either `napi_tsfn_blocking` to indicate that the call should block if the queue is full or `napi_tsfn_nonblocking` to indicate that the call should return immediately with a status of `napi_queue_full` whenever the queue is full.

This API should not be called with `napi_tsfn_blocking` from a JavaScript thread, because, if the queue is full, it may cause the JavaScript thread to deadlock.

This API will return `napi_closing` if `napi_release_threadsafe_function()` was called with `abort` set to `napi_tsfn_abort` from any thread. The value is only added to the queue if the API returns `napi_ok` .

This API may be called from any thread which makes use of `func` .

`napi_acquire_threadsafe_function`

```
NAPI_EXTERN napi_status
napi_acquire_threadsafe_function(napi_threadsafe_function func);
```

- `[in] func` : The asynchronous thread-safe JavaScript function to start making use of.

A thread should call this API before passing `func` to any other thread-safe function APIs to indicate that it will be making use of `func` . This prevents `func` from being destroyed when all other threads have stopped making use of it.

This API may be called from any thread which will start making use of `func`.

`napi_release_threadsafe_function`

```
NAPI_EXTERN napi_status
napi_release_threadsafe_function(napi_threadsafe_function func,
                                 napi_threadsafe_function_release_mode mode);
```

- `[in] func`: The asynchronous thread-safe JavaScript function whose reference count to decrement.
- `[in] mode`: Flag whose value can be either `napi_tsfn_release` to indicate that the current thread will make no further calls to the thread-safe function, or `napi_tsfn_abort` to indicate that in addition to the current thread, no other thread should make any further calls to the thread-safe function. If set to `napi_tsfn_abort`, further calls to `napi_call_threadsafe_function()` will return `napi_closing`, and no further values will be placed in the queue.

A thread should call this API when it stops making use of `func`. Passing `func` to any thread-safe APIs after having called this API has undefined results, as `func` may have been destroyed.

This API may be called from any thread which will stop making use of `func`.

`napi_ref_threadsafe_function`

```
NAPI_EXTERN napi_status
napi_ref_threadsafe_function(napi_env env, napi_threadsafe_function func);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] func`: The thread-safe function to reference.

This API is used to indicate that the event loop running on the main thread should not exit until `func` has been destroyed. Similar to `uv_ref` it is also idempotent.

Neither does `napi_unref_threadsafe_function` mark the thread-safe functions as able to be destroyed nor does `napi_ref_threadsafe_function` prevent it from being destroyed. `napi_acquire_threadsafe_function` and `napi_release_threadsafe_function` are available for that purpose.

This API may only be called from the main thread.

`napi_unref_threadsafe_function`

```
NAPI_EXTERN napi_status
napi_unref_threadsafe_function(napi_env env, napi_threadsafe_function func);
```

- `[in] env`: The environment that the API is invoked under.
- `[in] func`: The thread-safe function to unreference.

This API is used to indicate that the event loop running on the main thread may exit before `func` is destroyed. Similar to `uv_unref` it is also idempotent.

This API may only be called from the main thread.

Miscellaneous utilities

`node_api_get_module_file_name`

> *Stability: 1 - Experimental*

```
NAPI_EXTERN napi_status
node_api_get_module_file_name(napi_env env, const char** result);
```

- `[in] env` : The environment that the API is invoked under.
- `[out] result` : A URL containing the absolute path of the location from which the add-on was loaded. For a file on the local file system it will start with `file://` . The string is null-terminated and owned by `env` and must thus not be modified or freed.

`result` may be an empty string if the add-on loading process fails to establish the add-on's file name during loading.