# emscripten.h

This page documents the public C++ APIs provided by emscripten.h .

Emscripten uses existing/familiar APIs where possible (for example: SDL). This API provides C++ support for capabilities that are specific to JavaScript or the browser environment, or for which there is no existing API.

**Table of Contents**



## Inline assembly/JavaScript

Guide material for the following APIs can be found in Calling JavaScript from C/C++.

### Defines

`EM_ASM(...)`

Convenient syntax for inline assembly/JavaScript.

This allows you to declare JavaScript in your C code “inline”, which is then executed when your compiled code is run in the browser. For example, the following C code would display two alerts if it was compiled with Emscripten and run in the browser:

```
EM_ASM(alert('hai'); alert('bai'));
```

Arguments can be passed inside the JavaScript code block, where they arrive as variables `$0`, `$1` etc. These arguments can either be of type `int32_t` or `double`.

```
EM_ASM({
  console.log('I received: ' + [$0, $1]);
}, 100, 35.5);
```

Note the `{` and `}`.

Null-terminated C strings can also be passed into `EM_ASM` blocks, but to operate on them, they need to be copied out from the heap to convert to high-level JavaScript strings.

```
EM_ASM(console.log('hello ' + UTF8ToString($0)), "world!");
```

In the same manner, pointers to any type (including `void *`) can be passed inside `EM_ASM` code, where they appear as integers like `char *` pointers above did. Accessing the data can be managed by reading the heap directly.

```
        int arr[2] = { 30, 45 };
        EM_ASM({
          console.log('Data: ' + HEAP32[$0>>2] + ', ' + HEAP32[($0+4)>>2]);
        }, arr);

.. note::
        - As of Emscripten ``1.30.4``, the contents of ``EM_ASM`` code blocks appear inside the normal JS file, and as result, Closure compiler and other JavaScript minifiers will be able to operate on them. You may need to use safety quotes in some places (``a['b']`` instead of ``a.b``) to avoid minification fro occurring.
        - The C preprocessor does not have an understanding of JavaScript tokens, and as a result, if the ``code`` block contains a comma character ``,``, it may be necessary to wrap the code block inside parentheses. For example, code ``EM_ASM(return [1,2,3].length);`` will not compile, but ``EM_ASM((return [1,2,3].length));`` does.
```

`EM_ASM_INT(code, ...)` `EM_ASM_DOUBLE(code, ...)`

These two functions behave like EM_ASM, but in addition they also return a value back to C code. The output value is passed back with a `return` statement:

```
int x = EM_ASM_INT({
  return $0 + 42;
}, 100);

int y = EM_ASM_INT(return TOTAL_MEMORY);
```

Strings can be returned back to C from JavaScript, but one needs to be careful about memory management.

```
char *str = (char*)EM_ASM_INT({
        var jsString = 'Hello with some exotic Unicode characters: Tässä on yksi lumiukko: ☃, ole hyvä.';
        var lengthBytes = lengthBytesUTF8(jsString)+1; // 'jsString.length' would return the length of the string as UTF-16 units, but Emscripten C strings operate as UTF-8.
        var stringOnWasmHeap = _malloc(lengthBytes);
        stringToUTF8(jsString, stringOnWasmHeap, lengthBytes+1);
        return stringOnWasmHeap;
});
printf("UTF8 string says: %s\n", str);
free(str); // Each call to _malloc() must be paired with free(), or heap memory will leak!
```

# Calling JavaScript From C/C++

Guide material for the following APIs can be found in Calling JavaScript from C/C++.

## Function pointer types for callbacks

The following types are used to define function callback signatures used in a number of functions in this file.

`em_callback_func`

General function pointer type for use in callbacks with no parameters.

Defined as:

```
typedef void (*em_callback_func)(void)
```

### em_arg_callback_func

Generic function pointer type for use in callbacks with a single `void*` parameter.

This type is used to define function callbacks that need to pass arbitrary data. For example, *`emscripten_set_main_loop_arg()`* sets user-defined data, and passes it to a callback of this type on completion.

Defined as:

```
typedef void (*em_arg_callback_func)(void*)
```

### em_str_callback_func

General function pointer type for use in callbacks with a C string (`const char *`) parameter.

This type is used for function callbacks that need to be passed a C string. For example, it is used in *`emscripten_async_wget()`* to pass the name of a file that has been asynchronously loaded.

Defined as:

```
typedef void (*em_str_callback_func)(const char *)
```

## Functions

### void emscripten_run_script(const char *script)

Interface to the underlying JavaScript engine. This function will `eval()` the given script. Note: If -s NO_DYNAMIC_EXECUTION=1 is set, this function will not be available.

This function can be called from a pthread, and it is executed in the scope of the Web Worker that is hosting the pthread. To evaluate a function in the scope of the main runtime thread, see the function emscripten_sync_run_in_main_runtime_thread().

**Parameters:**
- **script** (*const char**) – The script to evaluate.

**Return type:** void

### int emscripten_run_script_int(const char *script)

Interface to the underlying JavaScript engine. This function will `eval()` the given script. Note: If -s NO_DYNAMIC_EXECUTION=1 is set, this function will not be available.

This function can be called from a pthread, and it is executed in the scope of the Web Worker that is hosting the pthread. To evaluate a function in the scope of the main runtime thread, see the function emscripten_sync_run_in_main_runtime_thread().

**Parameters:**
- **script** (*const char**) – The script to evaluate.

**Returns:** The result of the evaluation, as an integer.

**Return type:** int

### char * `emscripten_run_script_string` (const char *script)

Interface to the underlying JavaScript engine. This function will `eval()` the given script. Note that this overload uses a single buffer shared between calls. Note: If -s NO_DYNAMIC_EXECUTION=1 is set, this function will not be available.

This function can be called from a pthread, and it is executed in the scope of the Web Worker that is hosting the pthread. To evaluate a function in the scope of the main runtime thread, see the function emscripten_sync_run_in_main_runtime_thread().

**Parameters:**
- **script** (*const char**) – The script to evaluate.

**Returns:** The result of the evaluation, as a string.

**Return type:** char*

### void `emscripten_async_run_script` (const char *script, int millis)

Asynchronously run a script, after a specified amount of time.

This function can be called from a pthread, and it is executed in the scope of the Web Worker that is hosting the pthread. To evaluate a function in the scope of the main runtime thread, see the function emscripten_sync_run_in_main_runtime_thread().

**Parameters:**
- **script** (*const char**) – The script to evaluate.
- **millis** (*int*) – The amount of time before the script is run, in milliseconds.

**Return type:** void

**void** `emscripten_async_load_script` **(const char *_script_, em_callback_func _onload_, em_callback_func _onerror_)**

Asynchronously loads a script from a URL.

This integrates with the run dependencies system, so your script can call `addRunDependency` multiple times, prepare various asynchronous tasks, and call `removeRunDependency` on them; when all are complete (or if there were no run dependencies to begin with), `onload` is called. An example use for this is to load an asset module, that is, the output of the file packager.

This function is currently only available in main browser thread, and it will immediately fail by calling the supplied onerror() handler if called in a pthread.

**Parameters:**
- **script** (*const char**) – The script to evaluate.
- **onload** (*em_callback_func*) – A callback function, with no parameters, that is executed when the script has fully loaded.
- **onerror** (*em_callback_func*) – A callback function, with no parameters, that is executed if there is an error in loading.

**Return type:** void

# Browser Execution Environment

Guide material for the following APIs can be found in Emscripten Runtime Environment.

## Functions

**void** `emscripten_set_main_loop` **(em_callback_func _func_, int _fps_, int _simulate_infinite_loop_)**

Set a C function as the main event loop for the calling thread.

If the main loop function needs to receive user-defined data, use `emscripten_set_main_loop_arg()` instead.

The JavaScript environment will call that function at a specified number of frames per second. If called on the main browser thread, setting 0 or a negative value as the `fps` will use the browser's `requestAnimationFrame` mechanism to call the main loop function. This is **HIGHLY** recommended if you are doing rendering, as the browser's `requestAnimationFrame` will make sure you render at a proper smooth rate that lines up properly with the browser and monitor. If you do not render at all in your application, then you should pick a specific frame rate that makes sense for your code.

If `simulate_infinite_loop` is true, the function will throw an exception in order to stop execution of the caller. This will lead to the main loop being entered instead of code after the call to *`emscripten_set_main_loop()`* being run, which is the closest we can get to simulating an infinite loop (we do something similar in glutMainLoop in GLUT ). If this parameter is `false`, then the behavior is the same as it was before this parameter was added to the API, which is that execution continues normally. Note that in both cases we do not run global destructors, `atexit`, etc., since we know the main loop will still be running, but if we do not simulate an infinite loop then the stack will be unwound. That means that if `simulate_infinite_loop` is `false`, and you created an object on the stack, it will be cleaned up before the main loop is called for the first time.

This function can be called in a pthread, in which case the callback loop will be set up to be called in the context of the calling thread. In order for the loop to work, the calling thread must regularly “yield back” to the browser by exiting from its pthread main function, since the callback will be able to execute only when the calling thread is not executing any other code. This means that running a synchronously blocking main loop is not compatible with the emscripten_set_main_loop() function.

Since `requestAnimationFrame()` API is not available in web workers, when called `emscripten_set_main_loop()` in a pthread with `fps` <= 0, the effect of syncing up to the display’s refresh rate is emulated, and generally will not precisely line up with vsync intervals.

**Tip**

There can be only *one* main loop function at a time, per thread. To change the main loop function, first *`cancel`* the current loop, and then call this function to set another.

**Note**

See *`emscripten_set_main_loop_expected_blockers()`*, *`emscripten_pause_main_loop()`*, *`emscripten_resume_main_loop()`* and *`emscripten_cancel_main_loop()`* for information about blocking, pausing, and resuming the main loop of the calling thread.

**Note**

Calling this function overrides the effect of any previous calls to *`emscripten_set_main_loop_timing()`* in the calling thread by applying the timing mode specified by the parameter `fps`. To specify a different timing mode for the current thread, call the function *`emscripten_set_main_loop_timing()`* after setting up the main loop.

**Parameters:**

- **func** (*em_callback_func*) – C function to set as main event loop for the calling thread.
- **fps** (*int*) – Number of frames per second that the JavaScript will call the function. Setting `int <=0` (recommended) uses the browser's `requestAnimationFrame` mechanism to call the function.
- **simulate_infinite_loop** (*int*) – If true, this function will throw an exception in order to stop execution of the caller.

---

### void `emscripten_set_main_loop_arg` (em_arg_callback_func *func*, void **arg*, int *fps*, int *simulate_infinite_loop*)

Set a C function as the main event loop for the calling thread, passing it user-defined data.

**See also**

The information in *`emscripten_set_main_loop()`* also applies to this function.

**Parameters:**

- **func** (*em_arg_callback_func*) – C function to set as main event loop. The function signature must have a `void*` parameter for passing the `arg` value.
- **arg** (*void**) – User-defined data passed to the main loop function, untouched by the API itself.
- **fps** (*int*) – Number of frames per second at which the JavaScript will call the function. Setting `int <=0` (recommended) uses the browser's `requestAnimationFrame` mechanism to call the function.
- **simulate_infinite_loop** (*int*) – If true, this function will throw an exception in order to stop execution of the caller.

---

### void `emscripten_push_main_loop_blocker` (em_arg_callback_func *func*, void **arg*)

---

### void `emscripten_push_uncounted_main_loop_blocker` (em_arg_callback_func *func*, void **arg*)

Add a function that **blocks** the main loop for the calling thread.

The function is added to the back of a queue of events to be blocked; the main loop will not run until all blockers in the queue complete.

In the "counted" version, blockers are counted (internally) and `Module.setStatus` is called with some text to report progress (`setStatus` is a general hook that a program can define in order to show processing updates).

**Note**

- Main loop blockers block the main loop from running, and can be counted to show progress. In contrast, `emscripten_async_calls` are not counted, do not block the main loop, and can fire at specific time in the future.

**Parameters:**
- **func** (*em_arg_callback_func*) – The main loop blocker function. The function signature must have a `void*` parameter for passing the `arg` value.
- **arg** (*void**) – User-defined arguments to pass to the blocker function.

**Return type:** void

### void `emscripten_pause_main_loop`(void) void `emscripten_resume_main_loop`(void)

Pause and resume the main loop for the calling thread.

Pausing and resuming the main loop is useful if your app needs to perform some synchronous operation, for example to load a file from the network. It might be wrong to run the main loop before that finishes (the original code assumes that), so you can break the code up into asynchronous callbacks, but you must pause the main loop until they complete.

**Note**

These are fairly low-level functions. `emscripten_push_main_loop_blocker()` (and friends) provide more convenient alternatives.

### void `emscripten_cancel_main_loop`(void)

Cancels the main event loop for the calling thread.

See also `emscripten_set_main_loop()` and `emscripten_set_main_loop_arg()` for information about setting and using the main loop.

### int `emscripten_set_main_loop_timing`(int *mode*, int *value*)

Specifies the scheduling mode that the main loop tick function of the calling thread will be called with.

This function can be used to interactively control the rate at which Emscripten runtime drives the main loop specified by calling the function `emscripten_set_main_loop()`. In native development, this corresponds with the “swap interval” or the “presentation interval” for 3D rendering. The new tick interval specified by this function takes effect immediately on the existing main loop, and this function must be called only after setting up a main loop via `emscripten_set_main_loop()`.

**Parameters:**

- **mode** (*int*) –
  The timing mode to use. Allowed values are EM_TIMING_SETTIMEOUT, EM_TIMING_RAF and EM_TIMING_SETIMMEDIATE.

  **param int value:**

  The timing value to activate for the main loop. This value interpreted differently according to the `mode` parameter:

  - If `mode` is EM_TIMING_SETTIMEOUT, then `value` specifies the number of milliseconds to wait between subsequent ticks to the main loop and updates occur independent of the vsync rate of the display (vsync off). This method uses the JavaScript `setTimeout` function to drive the animation.
  - If `mode` is EM_TIMING_RAF, then updates are performed using the `requestAnimationFrame` function (with vsync enabled), and this value is interpreted as a “swap interval” rate for the main loop. The value of `1` specifies the runtime that it should render at every vsync (typically 60fps), whereas the value `2` means that the main loop callback should be called only every second vsync (30fps). As a general formula, the value `n` means that the main loop is updated at every n’th vsync, or at a rate of `60/n` for 60Hz displays, and `120/n` for 120Hz displays.
  - If `mode` is EM_TIMING_SETIMMEDIATE, then updates are performed using the `setImmediate` function, or if not available, emulated via `postMessage`. See *setImmediate on MDN <https://developer.mozilla.org/en-US/docs/Web/API/Window/setImmediate>* for more information. Note that this mode is **strongly not recommended** to be used when deploying Emscripten output to the web, since it depends on an unstable web extension that is in draft status, browsers other than IE do not currently support it, and its implementation has been considered controversial in review.

**rtype:** int

**return:** The value 0 is returned on success, and a nonzero value is returned on failure. A failure occurs if there is no main loop active before calling this function.

**Note**

Browsers heavily optimize towards using `requestAnimationFrame` for animation instead of the other provided modes. Because of that, for best experience across browsers, calling this function with `mode=EM_TIMING_RAF` and `value=1` will yield best results. Using the JavaScript `setTimeout` function is known to cause stutter and generally worse experience than using the `requestAnimationFrame` function.

**Note**

There is a functional difference between `setTimeout` and `requestAnimationFrame`: If the user minimizes the browser window or hides your application tab, browsers will typically stop calling `requestAnimationFrame` callbacks, but `setTimeout`-based main loop will continue to be run, although with heavily throttled intervals. See *setTimeout on MDN <https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers.setTimeout#Inactive_tabs>* for more information.

### void `emscripten_get_main_loop_timing`(int **mode*, int **value*)

Returns the current main loop timing mode that is in effect. For interpretation of the values, see the documentation of the function `emscripten_set_main_loop_timing()`. The timing mode is controlled by calling the functions `emscripten_set_main_loop_timing()` and `emscripten_set_main_loop()`.

**Parameters:**

- **mode** (*int**) – If not null, the used timing mode is returned here.
- **value** (*int**) – If not null, the used timing value is returned here.

### void `emscripten_set_main_loop_expected_blockers`(int *num*)

Sets the number of blockers that are about to be pushed.

The number is used for reporting the *relative progress* through a set of blockers, after which the main loop will continue.

For example, a game might have to run 10 blockers before starting a new level. The operation would first set this value as '10' and then push the 10 blockers. When the 3rd blocker (say) completes, progress is displayed as 3/10.

**Parameters:**
- **num** (*int*) – The number of blockers that are about to be pushed.

### void `emscripten_async_call`(em_arg_callback_func *func*, void **arg*, int *millis*)

Call a C function asynchronously, that is, after returning control to the JavaScript event loop.

This is done by a `setTimeout`.

When building natively this becomes a simple direct call, after `SDL_Delay` (you must include **SDL.h** for that).

If `millis` is negative, the browser's `requestAnimationFrame` mechanism is used.

**Parameters:**
- **func** (*em_arg_callback_func*) – The C function to call asynchronously. The function signature must have a `void*` parameter for passing the `arg` value.
- **arg** (*void**) – User-defined argument to pass to the C function.
- **millis** (*int*) – Timeout before function is called.

### void `emscripten_exit_with_live_runtime`(void)

Exits the program immediately, but leaves the runtime alive so that you can continue to run code later (so global destructors etc., are not run). Note that the runtime is kept alive automatically when you do an asynchronous operation like `emscripten_async_call()`, so you don't need to call this function for those cases.

### void `emscripten_force_exit`(int *status*)

Shuts down the runtime and exits (terminates) the program, as if you called `exit()`.

The difference is that `emscripten_force_exit` will shut down the runtime even if you previously called `emscripten_exit_with_live_runtime()` or otherwise kept the runtime alive. In other words, this method gives you the option to completely shut down the runtime after it was kept alive beyond the completion of `main()`.

Note that if `NO_EXIT_RUNTIME` is set (which it is by default) then the runtime cannot be shut down, as we do not include the code to do so. Build with `-s NO_EXIT_RUNTIME=0` if you want to be able to exit the runtime.

**Parameters:**
- **status** (*int*) – The same as for the *libc* function exit()

.

### double `emscripten_get_device_pixel_ratio` (void)

Returns the value of `window.devicePixelRatio`.

**Return type:** double

**Returns:** The pixel ratio or 1.0 if not supported.

### void `emscripten_hide_mouse` (void)

Hide the OS mouse cursor over the canvas.

Note that SDL's `SDL_ShowCursor` command shows and hides the SDL cursor, not the OS one. This command is useful to hide the OS cursor if your app draws its own cursor.

### void `emscripten_set_canvas_size` (int *width*, int *height*)

Resizes the pixel width and height of the `<canvas>` element on the Emscripten web page.

**Parameters:**
- **width** (*int*) – New pixel width of canvas element.
- **height** (*int*) – New pixel height of canvas element.

### void `emscripten_get_canvas_size` (int * *width*, int * *height*, int * *isFullscreen*)

Gets the current pixel width and height of the `<canvas>` element as well as whether the canvas is fullscreen or not.

**Parameters:**
- **width** (*int**) – Pixel width of canvas element.
- **height** (*int**) – New pixel height of canvas element.
- **isFullscreen** (*int**) – If True (`*int > 0`), `<canvas>` is full screen.

### double `emscripten_get_now` (void)

Returns the highest-precision representation of the current time that the browser provides.

This uses either `Date.now` or `performance.now`. The result is not an absolute time, and is only meaningful in comparison to other calls to this function.

**Return type:** double

**Returns:** The current time, in milliseconds (ms).

### float `emscripten_random`(void)

Generates a random number in the range 0-1. This maps to `Math.random()`.

**Return type:** float

**Returns:** A random number.

# Emscripten Asynchronous File System API

## Typedefs

### `em_async_wget_onload_func`

Function pointer type for the `onload` callback of *`emscripten_async_wget_data()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_async_wget_onload_func)(void*, void*, int)
```

### `em_async_wget2_onload_func`

Function pointer type for the `onload` callback of *`emscripten_async_wget2()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_async_wget2_onload_func)(void*, const char*)
```

### `em_async_wget2_onstatus_func`

Function pointer type for the `onerror` and `onprogress` callbacks of *`emscripten_async_wget2()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_async_wget2_onstatus_func)(void*, int)
```

### `em_async_wget2_data_onload_func`

Function pointer type for the `onload` callback of *`emscripten_async_wget2_data()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_async_wget2_data_onload_func)(void*, void *, unsigned*)
```

### em_async_wget2_data_onerror_func

Function pointer type for the `onerror` callback of *`emscripten_async_wget2_data()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_async_wget2_data_onerror_func)(void*, int, const char*)
```

### em_async_wget2_data_onprogress_func

Function pointer type for the `onprogress` callback of *`emscripten_async_wget2_data()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_async_wget2_data_onprogress_func)(void*, int, int)
```

### em_run_preload_plugins_data_onload_func

Function pointer type for the `onload` callback of *`emscripten_run_preload_plugins_data()`* (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_run_preload_plugins_data_onload_func)(void*, const char*)
```

## Functions

### void emscripten_wget (const char* *url*, const char* *file*)

Load file from url in *synchronously*. For the asynchronous version, see the *`emscripten_async_wget()`*.

In addition to fetching the URL from the network, preload plugins are executed so that the data is usable in `IMG_Load` and so forth (we synchronously do the work to make the browser decode the image or audio etc.).

This function is blocking; it won't return until all operations are finished. You can then open and read the file if it succeeded.

To use this function, you will need to compile your application with the linker flag `-s ASYNCIFY=1`

**Parameters:**

- **char* url** (*const*) – The URL to load.
- **char* file** (*const*) – The name of the file created and loaded from the URL. If the file already exists it will be overwritten. If the destination directory for the file does not exist on the filesystem, it will be created. A relative pathname may be passed, which will be interpreted relative to the current working directory at the time of the call to this function.

---

### void `emscripten_async_wget`(const char* *url*, const char* *file*, em_str_callback_func *onload*, em_str_callback_func *onerror*)

Loads a file from a URL asynchronously.

In addition to fetching the URL from the network, preload plugins are executed so that the data is usable in `IMG_Load` and so forth (we asynchronously do the work to make the browser decode the image or audio etc.).

When the file is ready the `onload` callback will be called. If any error occurs `onerror` will be called. The callbacks are called with the file as their argument.

**Parameters:**

- **char* url** (*const*) – The URL to load.
- **char* file** (*const*) – The name of the file created and loaded from the URL. If the file already exists it will be overwritten. If the destination directory for the file does not exist on the filesystem, it will be created. A relative pathname may be passed, which will be interpreted relative to the current working directory at the time of the call to this function.
- **onload** (*em_str_callback_func*) –
  Callback on successful load of the file. The callback function parameter value is:
  - *(const char)** : The name of the `file` that was loaded from the URL.
- **onerror** (*em_str_callback_func*) –
  Callback in the event of failure. The callback function parameter value is:
  - *(const char)** : The name of the `file` that failed to load from the URL.

**void** `emscripten_async_wget_data` **(const char* *url*, void **arg*, em_async_wget_onload_func *onload*, em_arg_callback_func *onerror*)**

Loads a buffer from a URL asynchronously.

This is the “data” version of `emscripten_async_wget()`.

Instead of writing to a file, this function writes to a buffer directly in memory. This avoids the overhead of using the emulated file system; note however that since files are not used, it cannot run preload plugins to set things up for `IMG_Load` and so forth (`IMG_Load` etc. work on files).

When the file is ready then the `onload` callback will be called. If any error occurred `onerror` will be called.

**Parameters:**

- **url** (*const char**) – The URL of the file to load.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **onload** (*em_async_wget_onload_func*) –
  Callback on successful load of the URL into the buffer. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(void)** : A pointer to a buffer with the data. Note that, as with the worker API, the data buffer only lives during the callback; it must be used or copied during that time.
  - *(int)* : The size of the buffer, in bytes.
- **onerror** (*em_arg_callback_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).

**int** `emscripten_async_wget2` **(const char* *url*, const char* *file*, const char* *requesttype*, const char* *param*, void **arg*, em_async_wget2_onload_func *onload*, em_async_wget2_onstatus_func *onerror*, em_async_wget2_onstatus_func *onprogress*)**

Loads a file from a URL asynchronously.

This is an **experimental** “more feature-complete” version of `emscripten_async_wget()`.

In addition to fetching the URL from the network, preload plugins are executed so that the data is usable in `IMG_Load` and so forth (we asynchronously do the work to make the browser decode the image, audio, etc.).

When the file is ready the `onload` callback will be called with the object pointers given in `arg` and `file`. During the download the `onprogress` callback is called.

**Parameters:**

- **url** (*const char**) – The URL of the file to load.
- **file** (*const char**) – The name of the file created and loaded from the URL. If the file already exists it will be overwritten. If the destination directory for the file does not exist on the filesystem, it will be created. A relative pathname may be passed, which will be interpreted relative to the current working directory at the time of the call to this function.
- **requesttype** (*const char**) – 'GET' or 'POST'.
- **param** (*const char**) – Request parameters for POST requests (see `requesttype`). The parameters are specified in the same way as they would be in the URL for an equivalent GET request: e.g. `key=value&key2=value2`.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **onload** (*em_async_wget2_onload_func*) –
  Callback on successful load of the file. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(const char)** : The `file` passed to the original call.
- **onerror** (*em_async_wget2_onstatus_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(int)* : The HTTP status code.
- **onprogress** (*em_async_wget2_onstatus_func*) –
  Callback during load of the file. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(int)* : The progress (percentage completed).

**Returns:** A handle to request (`int`) that can be used to `abort` the request.

---

**int `emscripten_async_wget2_data`(const char* *url*, const char* *requesttype*, const char* *param*, void **arg*, int *free*, em_async_wget2_data_onload_func *onload*, em_async_wget2_data_onerror_func *onerror*, em_async_wget2_data_onprogress_func *onprogress*)**

Loads a buffer from a URL asynchronously.

This is the "data" version of `emscripten_async_wget2()`. It is an **experimental** "more feature complete" version of `emscripten_async_wget_data()`.

Instead of writing to a file, this function writes to a buffer directly in memory. This avoids the overhead of using the emulated file system; note however that since files are not used, it cannot run preload plugins to set things up for `IMG_Load` and so forth (`IMG_Load` etc. work on files).

When the file is ready the `onload` callback will be called with the object pointers given in `arg`, a pointer to the buffer in memory, and an unsigned integer containing the size of the buffer. During the download the `onprogress` callback is called with progress information. If an error occurs, `onerror` will be called with the HTTP status code and a string containing the status description.

**Parameters:**

- **url** (*const char**) – The URL of the file to load.
- **requesttype** (*const char**) – 'GET' or 'POST'.
- **param** (*const char**) – Request parameters for POST requests (see `requesttype`). The parameters are specified in the same way as they would be in the URL for an equivalent GET request: e.g. `key=value&key2=value2`.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **free** (*int*) – Tells the runtime whether to free the returned buffer after `onload` is complete. If `false` freeing the buffer is the receiver's responsibility.
- **onload** (*em_async_wget2_data_onload_func*) –
  Callback on successful load of the file. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(void)** : A pointer to the buffer in memory.
  - *(unsigned)* : The size of the buffer (in bytes).
- **onerror** (*em_async_wget2_data_onerror_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(int)* : The HTTP error code.
  - *(const char)** : A string with the status description.
- **onprogress** (*em_async_wget2_data_onprogress_func*) –
  Callback called (regularly) during load of the file to update progress. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(int)* : The number of bytes loaded.
  - *(int)* : The total size of the data in bytes, or zero if the size is unavailable.

**Returns:** A handle to request (`int`) that can be used to `abort` the request.

**void `emscripten_async_wget2_abort` (int *handle*)**

Abort an asynchronous request raised using `emscripten_async_wget2()` or `emscripten_async_wget2_data()`.

**Parameters:**

- **handle** (*int*) – A handle to request to be aborted.

---

**void `emscripten_run_preload_plugins_data` (char* *data*, int *size*, const char **suffix*, void **arg*, em_run_preload_plugins_data_onload_func *onload*, em_arg_callback_func *onerror*)**

Runs preload plugins on a buffer of data asynchronously. This is a "data" version of `emscripten_run_preload_plugins()`, which receives raw data as input instead of a filename (this can prevent the need to write data to a file first).

When file is loaded then the `onload` callback will be called. If any error occurs `onerror` will be called.

`onload` also receives a second parameter, which is a 'fake' filename which you can pass into `IMG_Load` (it is not an actual file, but it identifies this image for `IMG_Load` to be able to process it). Note that the user of this API is responsible for `free()` ing the memory allocated for the fake filename.

**Parameters:**

- **data** (*char**) – The buffer of data to process.
- **suffix** (*const char**) – The file suffix, e.g. 'png' or 'jpg'.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **onload** (*em_run_preload_plugins_data_onload_func*) –
  Callback on successful processing of the data. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(const char)** : A 'fake' filename which you can pass into `IMG_Load`. See above for more information.
- **onerror** (*em_arg_callback_func*) –
  Callback in the event of failure. The callback function parameter value is:
  - *(void)** : Equal to `arg` (user defined data).

# Emscripten Asynchronous IndexedDB API

IndexedDB is a browser API that lets you store data persistently, that is, you can save data there and load it later when the user re-visits the web page. IDBFS provides one way to use IndexedDB, through the Emscripten filesystem layer. The `emscripten_idb_*` methods listed here provide an alternative API, directly to IndexedDB, thereby avoiding the overhead of the filesystem layer.

---

**void `emscripten_idb_async_load`(const char *db_name, const char *file_id, void* arg, em_async_wget_onload_func onload, em_arg_callback_func onerror)**

Loads data from local IndexedDB storage asynchronously. This allows use of persistent data, without the overhead of the filesystem layer.

When the data is ready then the `onload` callback will be called. If any error occurred `onerror` will be called.

**Parameters:**

- **db_name** – The IndexedDB database from which to load.
- **file_id** – The identifier of the data to load.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **onload** (*em_async_wget_onload_func*) –
  Callback on successful load of the URL into the buffer. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
  - *(void)** : A pointer to a buffer with the data. Note that, as with the worker API, the data buffer only lives during the callback; it must be used or copied during that time.
  - *(int)* : The size of the buffer, in bytes.
- **onerror** (*em_arg_callback_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).

---

`void emscripten_idb_async_store(const char *db_name, const char *file_id, void* ptr, int num, void* arg, em_arg`

Stores data to local IndexedDB storage asynchronously. This allows use of persistent data, without the overhead of the filesystem layer.

When the data has been stored then the `onstore` callback will be called. If any error occurred `onerror` will be called.

**Parameters:**

- **db_name** – The IndexedDB database from which to load.
- **file_id** – The identifier of the data to load.
- **ptr** – A pointer to the data to store.
- **num** – How many bytes to store.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **onstore** (*em_arg_callback_func*) –
  Callback on successful store of the data buffer to the URL. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).
- **onerror** (*em_arg_callback_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).

---

**void `emscripten_idb_async_delete`(const char *db_name, const char *file_id, void* arg, em_arg_callback_func ondelete, em_arg_callback_func onerror)**

Deletes data from local IndexedDB storage asynchronously.

When the data has been deleted then the `ondelete` callback will be called. If any error occurred `onerror` will be called.

**Parameters:**

- **db_name** – The IndexedDB database.
- **file_id** – The identifier of the data.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **ondelete** (*em_arg_callback_func*) –
  Callback on successful delete
  - *(void)** : Equal to `arg` (user defined data).
- **onerror** (*em_arg_callback_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).

---

**void `emscripten_idb_async_exists`(const char *db_name, const char *file_id, void* arg, em_idb_exists_func oncheck, em_arg_callback_func onerror)**

Checks if data with a certain ID exists in the local IndexedDB storage asynchronously.

When the data has been checked then the `oncheck` callback will be called. If any error occurred `onerror` will be called.

**Parameters:**

- **db_name** – The IndexedDB database.
- **file_id** – The identifier of the data.
- **arg** (*void**) – User-defined data that is passed to the callbacks, untouched by the API itself. This may be used by a callback to identify the associated call.
- **oncheck** (*em_idb_exists_func*) –
  Callback on successful check, with arguments
  - *(void)** : Equal to `arg` (user defined data).
  - *int* : Whether the file exists or not.
- **onerror** (*em_arg_callback_func*) –
  Callback in the event of failure. The callback function parameter values are:
  - *(void)** : Equal to `arg` (user defined data).

---

**int `emscripten_run_preload_plugins`(const char* *file*, em_str_callback_func *onload*, em_str_callback_func *onerror*)**

Runs preload plugins on a file asynchronously. It works on file data already present and performs any required asynchronous operations available as preload plugins, such as decoding images for use in `IMG_Load`, or decoding audio for use in `Mix_LoadWAV`.

Once the operations are complete, the `onload` callback will be called. If any error occurs `onerror` will be called. The callbacks are called with the file as their argument.

**Parameters:**

- **file** (*const char**) – The name of the file to process.
- **onload** (*em_str_callback_func*) –
  Callback on successful processing of the file. The callback function parameter value is:
  - *(const char)** : The name of the `file` that was processed.
- **onerror** (*em_str_callback_func*) –
  Callback in the event of failure. The callback function parameter value is:
  - *(const char)** : The name of the `file` for which the operation failed.

**Returns:** 0 if successful, -1 if the file does not exist

**Return type:** int

# Compiling

`EMSCRIPTEN_KEEPALIVE`

Forces LLVM to not dead-code-eliminate a function.

This also exports the function, as if you added it to EXPORTED_FUNCTIONS.

For example:

```
void EMSCRIPTEN_KEEPALIVE my_function() { printf("I am being kept alive\n"); }
```

# Worker API

## Typedefs

`int worker_handle`

A wrapper around web workers that lets you create workers and communicate with them.

Note that the current API is mainly focused on a main thread that sends jobs to workers and waits for responses, i.e., in an asymmetrical manner, there is no current API to send a message without being asked for it from a worker to the main thread.

`em_worker_callback_func`

Function pointer type for the callback from `emscripten_call_worker()` (specific values of the parameters documented in that method).

Defined as:

```
typedef void (*em_worker_callback_func)(char*, int, void*)
```

## Functions

**worker_handle `emscripten_create_worker` (const char * *url*)**

Creates a worker.

A worker must be compiled separately from the main program, and with the `BUILD_AS_WORKER` flag set to 1.

**Parameters:**
- **url** (*const char**) – The URL of the worker script.

**Returns:** A handle to the newly created worker.

**Return type:** worker_handle

---

**void `emscripten_destroy_worker`(worker_handle *worker*)**

Destroys a worker. See *`emscripten_create_worker()`*

**Parameters:**

- **worker** (*worker_handle*) – A handle to the worker to be destroyed.

---

**void `emscripten_call_worker`(worker_handle *worker*, const char **funcname*, char **data*, int *size*, em_worker_callback_func *callback*, void **arg*)**

Asynchronously calls a worker.

The worker function will be called with two parameters: a data pointer, and a size. The data block defined by the pointer and size exists only during the callback: **it cannot be relied upon afterwards**. If you need to keep some of that information outside the callback, then it needs to be copied to a safe location.

The called worker function can return data, by calling *`emscripten_worker_respond()`*. When the worker is called, if a callback was given it will be called with three arguments: a data pointer, a size, and an argument that was provided when calling *`emscripten_call_worker()`* (to more easily associate callbacks to calls). The data block defined by the data pointer and size behave like the data block in the worker function — it exists only during the callback.

**Parameters:**

- **worker** (*worker_handle*) – A handle to the worker to be called.
- **funcname** (*const char**) – The name of the function in the worker. The function must be a C function (so no C++ name mangling), and must be exported (EXPORTED_FUNCTIONS).
- **data** (*char**) – The address of a block of memory to copy over.
- **size** (*int*) – The size of the block of memory.
- **callback** (*em_worker_callback_func*) –
  Worker callback with the response. This can be `null`. The callback function parameter values are:
  - *(char)** : The `data` pointer provided in *`emscripten_call_worker()`*.
  - *(int)* : The `size` of the block of data.
  - *(void)** : Equal to `arg` (user defined data).
- **arg** (*void**) – An argument (user data) to be passed to the callback

void `emscripten_worker_respond`(char *data, int size)

void `emscripten_worker_respond_provisionally`(char *data, int size)

Sends a response when in a worker call (that is, when called by the main thread using `emscripten_call_worker()`).

Both functions post a message back to the thread which called the worker. The `emscripten_worker_respond_provisionally()` variant can be invoked multiple times, which will queue up messages to be posted to the worker's creator. Eventually, the _respond variant must be invoked, which will disallow further messages and free framework resources previously allocated for this worker call.

**Note**

Calling the provisional version is optional, but you must call the non-provisional version to avoid leaks.

**Parameters:**
- **data** (*char**) – The message to be posted.
- **size** (*int*) – The size of the message, in bytes.

int `emscripten_get_worker_queue_size`(worker_handle worker)

Checks how many responses are being waited for from a worker.

This only counts calls to `emscripten_call_worker()` that had a callback (calls with null callbacks are ignored), and where the response has not yet been received. It is a simple way to check on the status of the worker to see how busy it is, and do basic decisions about throttling.

**Parameters:**
- **worker** (*worker_handle*) – The handle to the relevant worker.

**Returns:** The number of responses waited on from a worker.

**Return type:** int

# Logging utilities

## Defines

`EM_LOG_CONSOLE`

If specified, logs directly to the browser console/inspector window. If not specified, logs via the application Module.

`EM_LOG_WARN`

If specified, prints a warning message.

`EM_LOG_ERROR`

If specified, prints an error message. If neither `EM_LOG_WARN` or `EM_LOG_ERROR` is specified, an info message is printed. `EM_LOG_WARN` and `EM_LOG_ERROR` are mutually exclusive.

`EM_LOG_C_STACK`

If specified, prints a call stack that contains file names referring to original C sources using source map information.

`EM_LOG_JS_STACK`

If specified, prints a call stack that contains file names referring to lines in the built .js/.html file along with the message. The flags `EM_LOG_C_STACK` and `EM_LOG_JS_STACK` can be combined to output both untranslated and translated file and line information.

`EM_LOG_DEMANGLE`

If specified, C/C++ function names are de-mangled before printing. Otherwise, the mangled post-compilation JavaScript function names are displayed.

`EM_LOG_NO_PATHS`

If specified, the pathnames of the file information in the call stack will be omitted.

`EM_LOG_FUNC_PARAMS`

If specified, prints out the actual values of the parameters the functions were invoked with.

## Functions

**int** `emscripten_get_compiler_setting` **(const char *_name_)**

Returns the value of a compiler setting.

For example, to return the integer representing the value of `PRECISE_F32` during compilation:

```
emscripten_get_compiler_setting("PRECISE_F32")
```

For values containing anything other than an integer, a string is returned (you will need to cast the `int` return value to a `char*`).

Some useful things this can do is provide the version of Emscripten ("EMSCRIPTEN_VERSION"), the optimization level ("OPT_LEVEL"), debug level ("DEBUG_LEVEL"), etc.

For this command to work, you must build with the following compiler option (as we do not want to increase the build size with this metadata):

```
-s RETAIN_COMPILER_SETTINGS=1
```

**Parameters:**
- **name** (*const char**) – The compiler setting to return.

**Returns:** The value of the specified setting. Note that for values other than an integer, a string is returned (cast the `int` return value to a `char*`).

**Return type:** int

### void `emscripten_debugger` ()

Emits `debugger`.

This is inline in the code, which tells the JavaScript engine to invoke the debugger if it gets there.

### void `emscripten_log` (int *flags*, ...)

Prints out a message to the console, optionally with the callstack information.

**Parameters:**
- **flags** (*int*) – A binary OR of items from the list of `EM_LOG_xxx` flags that specify printing options.
- **...** – A `printf`-style "format, ..." parameter list that is parsed according to the `printf` formatting rules.

### int `emscripten_get_callstack` (int *flags*, char **out*, int *maxbytes*)

Programmatically obtains the current callstack.

To query the amount of bytes needed for a callstack without writing it, pass 0 to `out` and `maxbytes`, in which case the function will return the number of bytes (including the terminating zero) that will be needed to hold the full callstack. Note that this might be fully accurate since subsequent calls will carry different line numbers, so it is best to allocate a few bytes extra to be safe.

**Parameters:**
- **flags** (*int*) – A binary OR of items from the list of `EM_LOG_xxx` flags that specify printing options. The flags `EM_LOG_CONSOLE`, `EM_LOG_WARN` and `EM_LOG_ERROR` do not apply in this function and are ignored.
- **out** (*char**) – A pointer to a memory region where the callstack string will be written to. The string outputted by this function will always be null-terminated.
- **maxbytes** (*int*) – The maximum number of bytes that this function can write to the memory pointed to by `out`. If there is not enough space, the output will be truncated (but always null-terminated).

**Returns:** The number of bytes written (not number of characters, so this will also include the terminating zero).

**Return type:** int

---

### char * `emscripten_get_preloaded_image_data` (const char *path, int *w, int *h)

Gets preloaded image data and the size of the image.

The function returns pointer to loaded image or NULL — the pointer should be `free()`'d. The width/height of the image are written to the `w` and `h` parameters if the data is valid.

**Parameters:**
- **path** – Full path/filename to the file containing the preloaded image.
- **w** (*int**) – Width of the image (if data is valid).
- **h** (*int**) – Height of the image (if data is valid).

**Type:** const char*

**Returns:** A pointer to the preloaded image or NULL.

**Return type:** char*

---

### char * `emscripten_get_preloaded_image_data_from_FILE` (FILE *file, int *w, int *h)

Gets preloaded image data from a C `FILE*`.

**Parameters:**
- **file** (*FILE**) – The `FILE` containing the preloaded image.
- **w** (*int**) – Width of the image (if data is valid).
- **h** (*int**) – Height of the image (if data is valid).

**Type:** const char*

**Returns:** A pointer to the preloaded image or NULL.

**Return type:** char*

---

**int `emscripten_print_double`(double *x*, char **to*, signed *max*)**

Prints a double as a string, including a null terminator. This is useful because JS engines have good support for printing out a double in a way that takes the least possible size, but preserves all the information in the double, i.e., it can then be parsed back in a perfectly reversible manner (snprintf etc. do not do so, sadly).

**Parameters:**
- **x** (*double*) – The double.
- **to** (*char**) – A pre-allocated buffer of sufficient size, or NULL if no output is requested (useful to get the necessary size).
- **max** (*signed*) – The maximum number of bytes that can be written to the output pointer 'to' (including the null terminator).

**Return type:** The number of necessary bytes, not including the null terminator (actually written, if `to` is not NULL).

## Socket event registration

The functions in this section register callback functions for receiving socket events. These events are analogous to WebSocket events but are emitted *after* the internal Emscripten socket processing has occurred. This means, for example, that the message callback will be triggered after the data has been added to the *recv_queue*, so that an application receiving this callback can simply read the data using the file descriptor passed as a parameter to the callback. All of the callbacks are passed a file descriptor (`fd`) representing the socket that the notified activity took place on. The error callback also takes an `int` representing the socket error number (`errno`) and a `char*` that represents the error message (`msg`).

Only a single callback function may be registered to handle any given event, so calling a given registration function more than once will cause the first callback to be replaced. Similarly, passing a `NULL` callback function to any `emscripten_set_socket_*_callback` call will de-register the callback registered for that event.

The `userData` pointer allows arbitrary data specified during event registration to be passed to the callback, this is particularly useful for passing `this` pointers around in Object Oriented code.

In addition to being able to register network callbacks from C it is also possible for native JavaScript code to directly use the underlying mechanism used to implement the callback registration. For example, the following code shows simple logging callbacks that are registered by default when `SOCKET_DEBUG` is enabled:

```
Module['websocket']['on']('error', function(error) {console.log('Socket error ' + error);});
Module['websocket']['on']('open', function(fd) {console.log('Socket open fd = ' + fd);});
Module['websocket']['on']('listen', function(fd) {console.log('Socket listen fd = ' + fd);});
Module['websocket']['on']('connection', function(fd) {console.log('Socket connection fd = ' + fd);});
Module['websocket']['on']('message', function(fd) {console.log('Socket message fd = ' + fd);});
Module['websocket']['on']('close', function(fd) {console.log('Socket close fd = ' + fd);});
```

Most of the JavaScript callback functions above get passed the file descriptor of the socket that triggered the callback, the on error callback however gets passed an *array* that contains the file descriptor, the error code and an error message.

**Note**

The underlying JavaScript implementation doesn't pass `userData`. This is mostly of use to C/C++ code and the `emscripten_set_socket_*_callback` calls simply create a closure containing the `userData` and pass that as the callback to the underlying JavaScript event registration mechanism.

## Callback functions

`em_socket_callback`

Function pointer for *`emscripten_set_socket_open_callback()`*, and the other socket functions (except *`emscripten_set_socket_error_callback()`*). This is defined as:

```
typedef void (*em_socket_callback)(int fd, void *userData);
```

**Parameters:**

- **fd** (*int*) – The file descriptor of the socket that triggered the callback.
- **userData** (*void**) – The `userData` originally passed to the event registration function.

### em_socket_error_callback

Function pointer for the *emscripten_set_socket_error_callback()*, defined as:

```
typedef void (*em_socket_error_callback)(int fd, int err, const char* msg, void *userData);
```

**Parameters:**

- **fd** (*int*) – The file descriptor of the socket that triggered the callback.
- **err** (*int*) – The code for the error that occurred.
- **msg** (*int*) – The message for the error that occurred.
- **userData** (*void**) – The `userData` originally passed to the event registration function.

## Functions

### void emscripten_set_socket_error_callback(void *userData, em_socket_error_callback callback)

Triggered by a `WebSocket` error.

See Socket event registration for more information.

**Parameters:**

- **userData** (*void**) – Arbitrary user data to be passed to the callback.
- **callback** (*em_socket_error_callback*) – Pointer to a callback function. The callback returns a file descriptor, error code and message, and the arbitrary `userData` passed to this function.

### void emscripten_set_socket_open_callback(void *userData, em_socket_callback callback)

Triggered when the `WebSocket` has opened.

See Socket event registration for more information.

**Parameters:**

- **userData** (*void**) – Arbitrary user data to be passed to the callback.
- **callback** (*em_socket_callback*) – Pointer to a callback function. The callback returns a file descriptor and the arbitrary `userData` passed to this function.

### void emscripten_set_socket_listen_callback(void *userData, em_socket_callback callback)

Triggered when `listen` has been called (synthetic event).

See Socket event registration for more information.

**Parameters:**
- **userData** (*void**) – Arbitrary user data to be passed to the callback.
- **callback** (*em_socket_callback*) – Pointer to a callback function. The callback returns a file descriptor and the arbitrary `userData` passed to this function.

---

**void `emscripten_set_socket_connection_callback`(void **userData*, em_socket_callback *callback*)**

Triggered when the connection has been established.

See Socket event registration for more information.

**Parameters:**
- **userData** (*void**) – Arbitrary user data to be passed to the callback.
- **callback** (*em_socket_callback*) – Pointer to a callback function. The callback returns a file descriptor and the arbitrary `userData` passed to this function.

---

**void `emscripten_set_socket_message_callback`(void **userData*, em_socket_callback *callback*)**

Triggered when data is available to be read from the socket.

See Socket event registration for more information.

**Parameters:**
- **userData** (*void**) – Arbitrary user data to be passed to the callback.
- **callback** (*em_socket_callback*) – Pointer to a callback function. The callback returns a file descriptor and the arbitrary `userData` passed to this function.

---

**void `emscripten_set_socket_close_callback`(void **userData*, em_socket_callback *callback*)**

Triggered when the `WebSocket` has closed.

See Socket event registration for more information.

**Parameters:**
- **userData** (*void**) – Arbitrary user data to be passed to the callback.
- **callback** (*em_socket_callback*) – Pointer to a callback function. The callback returns a file descriptor and the arbitrary `userData` passed to this function.

# Unaligned types

## Typedefs

---

`emscripten_align1_short` `emscripten_align2_int` `emscripten_align1_int`

`emscripten_align2_float` `emscripten_align1_float` `emscripten_align4_double`

`emscripten_align2_double` `emscripten_align1_double`

Unaligned types. These may be used to force LLVM to emit unaligned loads/stores in places in your code where SAFE_HEAP found an unaligned operation.

For usage examples see tests/core/test_set_align.c .

**Note**

It is better to avoid unaligned operations, but if you are reading from a packed stream of bytes or such, these types may be useful!

# Emterpreter-Async functions

Emterpreter-async functions are asynchronous functions that appear synchronously in C, the linker flags `-s EMTERPRETIFY=1 -s EMTERPRETIFY_ASYNC=1` are required to use these functions. See Emterpreter for more details.

## Sleeping

**void `emscripten_sleep`(unsigned int *ms*)**

Sleep for *ms* milliseconds. This is a normal "synchronous" sleep, which blocks all other operations while it runs. In other words, if there are other async events waiting to happen, they will not happen during this sleep, which makes sense as conceptually this code is on the stack (that's how it looks in the C source code). If you do want things to happen while sleeping, see `emscripten_sleep_with_yield`.

**void `emscripten_sleep_with_yield`(unsigned int *ms*)**

Sleep for *ms* milliseconds, while allowing other asynchronous operations, e.g. caused by `emscripten_async_call`, to run normally, during this sleep. Note that this method **does** still block the main loop, as otherwise it could recurse, if you are calling this method from it. Even so, you should use this method carefully: the order of execution is potentially very confusing this way.

## Network

```
void emscripten_wget_data(const char* url, void** pbuffer, int* pnum, int *perror);
```

Synchronously fetches data off the network, and stores it to a buffer in memory, which is allocated for you. **You must free the buffer, or it will leak!**

**Parameters:**

- **url** – The URL to fetch from
- **pbuffer** – An out parameter that will be filled with a pointer to a buffer containing the data that is downloaded. This space has been malloced for you, **and you must free it, or it will leak!**
- **pnum** – An out parameter that will be filled with the size of the downloaded data.
- **perror** – An out parameter that will be filled with a non-zero value if an error occurred.

## IndexedDB

```
void emscripten_idb_load(const char *db_name, const char *file_id, void** pbuffer, int* pnum, int *perror);
```

Synchronously fetches data from IndexedDB, and stores it to a buffer in memory, which is allocated for you. **You must free the buffer, or it will leak!**

**Parameters:**

- **db_name** – The name of the database to load from
- **file_id** – The name of the file to load
- **pbuffer** – An out parameter that will be filled with a pointer to a buffer containing the data that is downloaded. This space has been malloced for you, **and you must free it, or it will leak!**
- **pnum** – An out parameter that will be filled with the size of the downloaded data.
- **perror** – An out parameter that will be filled with a non-zero value if an error occurred.

```
void emscripten_idb_store(const char *db_name, const char *file_id, void* buffer, int num, int *perror);
```

Synchronously stores data to IndexedDB.

**Parameters:**

- **db_name** – The name of the database to store to
- **file_id** – The name of the file to store
- **buffer** – A pointer to the data to store
- **num** – How many bytes to store
- **perror** – An out parameter that will be filled with a non-zero value if an error occurred.

```
void emscripten_idb_delete(const char *db_name, const char *file_id, int *perror);
```

Synchronously deletes data from IndexedDB.

**Parameters:**

- **db_name** – The name of the database to delete from
- **file_id** – The name of the file to delete
- **perror** – An out parameter that will be filled with a non-zero value if an error occurred.

---

```
void emscripten_idb_exists(const char *db_name, const char *file_id, int* pexists, int *perror);
```

Synchronously checks if a file exists in IndexedDB.

**Parameters:**

- **db_name** – The name of the database to check in
- **file_id** – The name of the file to check
- **pexists** – An out parameter that will be filled with a non-zero value if the file exists in that database.
- **perror** – An out parameter that will be filled with a non-zero value if an error occurred.

# Asyncify functions

Asyncify functions are asynchronous functions that appear synchronously in C, the linker flag *-s ASYNCIFY=1* is required to use these functions. See Asyncify for more details.

## Typedefs

---

`emscripten_coroutine`

A handle to the structure used by coroutine supporting functions.

## Functions

---

**void** `emscripten_sleep` **(unsigned int *ms*)**

Sleep for *ms* milliseconds.

---

**emscripten_coroutine** `emscripten_coroutine_create` **(em_arg_callback_func *func*, void **arg*, int *stack_size*)**

Create a coroutine which will be run as *func(arg)*.

**Parameters:**

- **stack_size** (*int*) – the stack size that should be allocated for the coroutine, use 0 for the default value.

### int `emscripten_coroutine_next`(emscripten_coroutine *coroutine*)

Run *coroutine* until it returns, or *emscripten_yield* is called. A non-zero value is returned if *emscripten_yield* is called, otherwise 0 is returned, and future calls of *emscripten_coroutine_next* on this coroutine is undefined behaviour.

### void `emscripten_yield`(void)

This function should only be called in a coroutine created by *emscripten_coroutine_create*, when it called, the coroutine is paused and the caller will continue.

Previous Next

Report Bug Licensing Contributing Mailing list Wiki Release notes Blogs Contact