/\*\*

\* The `fs` module enables interacting with the file system in a

\* way modeled on standard POSIX functions.

\*

\* To use the promise-based APIs:

\*

\* ```js

\* import \* as fs from 'fs/promises';

\* ```

\*

\* To use the callback and sync APIs:

\*

\* ```js

\* import \* as fs from 'fs';

\* ```

\*

\* All file system operations have synchronous, callback, and promise-based

\* forms, and are accessible using both CommonJS syntax and ES6 Modules (ESM).

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/fs.js)

\*/

declare module 'fs' {

import \* as stream from 'node:stream';

import { Abortable, EventEmitter } from 'node:events';

import { URL } from 'node:url';

import \* as promises from 'node:fs/promises';

export { promises };

/\*\*

\* Valid types for path values in "fs".

\*/

export type PathLike = string | Buffer | URL;

export type PathOrFileDescriptor = PathLike | number;

export type TimeLike = string | number | Date;

export type NoParamCallback = (err: NodeJS.ErrnoException | null) => void;

export type BufferEncodingOption =

| 'buffer'

| {

encoding: 'buffer';

};

export interface ObjectEncodingOptions {

encoding?: BufferEncoding | null | undefined;

}

export type EncodingOption = ObjectEncodingOptions | BufferEncoding | undefined | null;

export type OpenMode = number | string;

export type Mode = number | string;

export interface StatsBase<T> {

isFile(): boolean;

isDirectory(): boolean;

isBlockDevice(): boolean;

isCharacterDevice(): boolean;

isSymbolicLink(): boolean;

isFIFO(): boolean;

isSocket(): boolean;

dev: T;

ino: T;

mode: T;

nlink: T;

uid: T;

gid: T;

rdev: T;

size: T;

blksize: T;

blocks: T;

atimeMs: T;

mtimeMs: T;

ctimeMs: T;

birthtimeMs: T;

atime: Date;

mtime: Date;

ctime: Date;

birthtime: Date;

}

export interface Stats extends StatsBase<number> {}

/\*\*

\* A `fs.Stats` object provides information about a file.

\*

\* Objects returned from {@link stat}, {@link lstat} and {@link fstat} and

\* their synchronous counterparts are of this type.

\* If `bigint` in the `options` passed to those methods is true, the numeric values

\* will be `bigint` instead of `number`, and the object will contain additional

\* nanosecond-precision properties suffixed with `Ns`.

\*

\* ```console

\* Stats {

\* dev: 2114,

\* ino: 48064969,

\* mode: 33188,

\* nlink: 1,

\* uid: 85,

\* gid: 100,

\* rdev: 0,

\* size: 527,

\* blksize: 4096,

\* blocks: 8,

\* atimeMs: 1318289051000.1,

\* mtimeMs: 1318289051000.1,

\* ctimeMs: 1318289051000.1,

\* birthtimeMs: 1318289051000.1,

\* atime: Mon, 10 Oct 2011 23:24:11 GMT,

\* mtime: Mon, 10 Oct 2011 23:24:11 GMT,

\* ctime: Mon, 10 Oct 2011 23:24:11 GMT,

\* birthtime: Mon, 10 Oct 2011 23:24:11 GMT }

\* ```

\*

\* `bigint` version:

\*

\* ```console

\* BigIntStats {

\* dev: 2114n,

\* ino: 48064969n,

\* mode: 33188n,

\* nlink: 1n,

\* uid: 85n,

\* gid: 100n,

\* rdev: 0n,

\* size: 527n,

\* blksize: 4096n,

\* blocks: 8n,

\* atimeMs: 1318289051000n,

\* mtimeMs: 1318289051000n,

\* ctimeMs: 1318289051000n,

\* birthtimeMs: 1318289051000n,

\* atimeNs: 1318289051000000000n,

\* mtimeNs: 1318289051000000000n,

\* ctimeNs: 1318289051000000000n,

\* birthtimeNs: 1318289051000000000n,

\* atime: Mon, 10 Oct 2011 23:24:11 GMT,

\* mtime: Mon, 10 Oct 2011 23:24:11 GMT,

\* ctime: Mon, 10 Oct 2011 23:24:11 GMT,

\* birthtime: Mon, 10 Oct 2011 23:24:11 GMT }

\* ```

\* @since v0.1.21

\*/

export class Stats {}

/\*\*

\* A representation of a directory entry, which can be a file or a subdirectory

\* within the directory, as returned by reading from an `fs.Dir`. The

\* directory entry is a combination of the file name and file type pairs.

\*

\* Additionally, when {@link readdir} or {@link readdirSync} is called with

\* the `withFileTypes` option set to `true`, the resulting array is filled with `fs.Dirent` objects, rather than strings or `Buffer` s.

\* @since v10.10.0

\*/

export class Dirent {

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a regular file.

\* @since v10.10.0

\*/

isFile(): boolean;

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a file system

\* directory.

\* @since v10.10.0

\*/

isDirectory(): boolean;

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a block device.

\* @since v10.10.0

\*/

isBlockDevice(): boolean;

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a character device.

\* @since v10.10.0

\*/

isCharacterDevice(): boolean;

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a symbolic link.

\* @since v10.10.0

\*/

isSymbolicLink(): boolean;

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a first-in-first-out

\* (FIFO) pipe.

\* @since v10.10.0

\*/

isFIFO(): boolean;

/\*\*

\* Returns `true` if the `fs.Dirent` object describes a socket.

\* @since v10.10.0

\*/

isSocket(): boolean;

/\*\*

\* The file name that this `fs.Dirent` object refers to. The type of this

\* value is determined by the `options.encoding` passed to {@link readdir} or {@link readdirSync}.

\* @since v10.10.0

\*/

name: string;

}

/\*\*

\* A class representing a directory stream.

\*

\* Created by {@link opendir}, {@link opendirSync}, or `fsPromises.opendir()`.

\*

\* ```js

\* import { opendir } from 'fs/promises';

\*

\* try {

\* const dir = await opendir('./');

\* for await (const dirent of dir)

\* console.log(dirent.name);

\* } catch (err) {

\* console.error(err);

\* }

\* ```

\*

\* When using the async iterator, the `fs.Dir` object will be automatically

\* closed after the iterator exits.

\* @since v12.12.0

\*/

export class Dir implements AsyncIterable<Dirent> {

/\*\*

\* The read-only path of this directory as was provided to {@link opendir},{@link opendirSync}, or `fsPromises.opendir()`.

\* @since v12.12.0

\*/

readonly path: string;

/\*\*

\* Asynchronously iterates over the directory via `readdir(3)` until all entries have been read.

\*/

[Symbol.asyncIterator](): AsyncIterableIterator<Dirent>;

/\*\*

\* Asynchronously close the directory's underlying resource handle.

\* Subsequent reads will result in errors.

\*

\* A promise is returned that will be resolved after the resource has been

\* closed.

\* @since v12.12.0

\*/

close(): Promise<void>;

close(cb: NoParamCallback): void;

/\*\*

\* Synchronously close the directory's underlying resource handle.

\* Subsequent reads will result in errors.

\* @since v12.12.0

\*/

closeSync(): void;

/\*\*

\* Asynchronously read the next directory entry via [`readdir(3)`](http://man7.org/linux/man-pages/man3/readdir.3.html) as an `fs.Dirent`.

\*

\* A promise is returned that will be resolved with an `fs.Dirent`, or `null`if there are no more directory entries to read.

\*

\* Directory entries returned by this function are in no particular order as

\* provided by the operating system's underlying directory mechanisms.

\* Entries added or removed while iterating over the directory might not be

\* included in the iteration results.

\* @since v12.12.0

\* @return containing {fs.Dirent|null}

\*/

read(): Promise<Dirent | null>;

read(cb: (err: NodeJS.ErrnoException | null, dirEnt: Dirent | null) => void): void;

/\*\*

\* Synchronously read the next directory entry as an `fs.Dirent`. See the

\* POSIX [`readdir(3)`](http://man7.org/linux/man-pages/man3/readdir.3.html) documentation for more detail.

\*

\* If there are no more directory entries to read, `null` will be returned.

\*

\* Directory entries returned by this function are in no particular order as

\* provided by the operating system's underlying directory mechanisms.

\* Entries added or removed while iterating over the directory might not be

\* included in the iteration results.

\* @since v12.12.0

\*/

readSync(): Dirent | null;

}

/\*\*

\* Class: fs.StatWatcher

\* @since v14.3.0, v12.20.0

\* Extends `EventEmitter`

\* A successful call to {@link watchFile} method will return a new fs.StatWatcher object.

\*/

export interface StatWatcher extends EventEmitter {

/\*\*

\* When called, requests that the Node.js event loop \_not\_ exit so long as the `fs.StatWatcher` is active. Calling `watcher.ref()` multiple times will have

\* no effect.

\*

\* By default, all `fs.StatWatcher` objects are "ref'ed", making it normally

\* unnecessary to call `watcher.ref()` unless `watcher.unref()` had been

\* called previously.

\* @since v14.3.0, v12.20.0

\*/

ref(): this;

/\*\*

\* When called, the active `fs.StatWatcher` object will not require the Node.js

\* event loop to remain active. If there is no other activity keeping the

\* event loop running, the process may exit before the `fs.StatWatcher` object's

\* callback is invoked. Calling `watcher.unref()` multiple times will have

\* no effect.

\* @since v14.3.0, v12.20.0

\*/

unref(): this;

}

export interface FSWatcher extends EventEmitter {

/\*\*

\* Stop watching for changes on the given `fs.FSWatcher`. Once stopped, the `fs.FSWatcher` object is no longer usable.

\* @since v0.5.8

\*/

close(): void;

/\*\*

\* events.EventEmitter

\* 1. change

\* 2. error

\*/

addListener(event: string, listener: (...args: any[]) => void): this;

addListener(event: 'change', listener: (eventType: string, filename: string | Buffer) => void): this;

addListener(event: 'error', listener: (error: Error) => void): this;

addListener(event: 'close', listener: () => void): this;

on(event: string, listener: (...args: any[]) => void): this;

on(event: 'change', listener: (eventType: string, filename: string | Buffer) => void): this;

on(event: 'error', listener: (error: Error) => void): this;

on(event: 'close', listener: () => void): this;

once(event: string, listener: (...args: any[]) => void): this;

once(event: 'change', listener: (eventType: string, filename: string | Buffer) => void): this;

once(event: 'error', listener: (error: Error) => void): this;

once(event: 'close', listener: () => void): this;

prependListener(event: string, listener: (...args: any[]) => void): this;

prependListener(event: 'change', listener: (eventType: string, filename: string | Buffer) => void): this;

prependListener(event: 'error', listener: (error: Error) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependOnceListener(event: string, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'change', listener: (eventType: string, filename: string | Buffer) => void): this;

prependOnceListener(event: 'error', listener: (error: Error) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

}

/\*\*

\* Instances of `fs.ReadStream` are created and returned using the {@link createReadStream} function.

\* @since v0.1.93

\*/

export class ReadStream extends stream.Readable {

close(callback?: (err?: NodeJS.ErrnoException | null) => void): void;

/\*\*

\* The number of bytes that have been read so far.

\* @since v6.4.0

\*/

bytesRead: number;

/\*\*

\* The path to the file the stream is reading from as specified in the first

\* argument to `fs.createReadStream()`. If `path` is passed as a string, then`readStream.path` will be a string. If `path` is passed as a `Buffer`, then`readStream.path` will be a

\* `Buffer`. If `fd` is specified, then`readStream.path` will be `undefined`.

\* @since v0.1.93

\*/

path: string | Buffer;

/\*\*

\* This property is `true` if the underlying file has not been opened yet,

\* i.e. before the `'ready'` event is emitted.

\* @since v11.2.0, v10.16.0

\*/

pending: boolean;

/\*\*

\* events.EventEmitter

\* 1. open

\* 2. close

\* 3. ready

\*/

addListener(event: 'close', listener: () => void): this;

addListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

addListener(event: 'end', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'open', listener: (fd: number) => void): this;

addListener(event: 'pause', listener: () => void): this;

addListener(event: 'readable', listener: () => void): this;

addListener(event: 'ready', listener: () => void): this;

addListener(event: 'resume', listener: () => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

on(event: 'close', listener: () => void): this;

on(event: 'data', listener: (chunk: Buffer | string) => void): this;

on(event: 'end', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'open', listener: (fd: number) => void): this;

on(event: 'pause', listener: () => void): this;

on(event: 'readable', listener: () => void): this;

on(event: 'ready', listener: () => void): this;

on(event: 'resume', listener: () => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'data', listener: (chunk: Buffer | string) => void): this;

once(event: 'end', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'open', listener: (fd: number) => void): this;

once(event: 'pause', listener: () => void): this;

once(event: 'readable', listener: () => void): this;

once(event: 'ready', listener: () => void): this;

once(event: 'resume', listener: () => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

prependListener(event: 'end', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'open', listener: (fd: number) => void): this;

prependListener(event: 'pause', listener: () => void): this;

prependListener(event: 'readable', listener: () => void): this;

prependListener(event: 'ready', listener: () => void): this;

prependListener(event: 'resume', listener: () => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

prependOnceListener(event: 'end', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'open', listener: (fd: number) => void): this;

prependOnceListener(event: 'pause', listener: () => void): this;

prependOnceListener(event: 'readable', listener: () => void): this;

prependOnceListener(event: 'ready', listener: () => void): this;

prependOnceListener(event: 'resume', listener: () => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

/\*\*

\* \* Extends `stream.Writable`

\*

\* Instances of `fs.WriteStream` are created and returned using the {@link createWriteStream} function.

\* @since v0.1.93

\*/

export class WriteStream extends stream.Writable {

/\*\*

\* Closes `writeStream`. Optionally accepts a

\* callback that will be executed once the `writeStream`is closed.

\* @since v0.9.4

\*/

close(callback?: (err?: NodeJS.ErrnoException | null) => void): void;

/\*\*

\* The number of bytes written so far. Does not include data that is still queued

\* for writing.

\* @since v0.4.7

\*/

bytesWritten: number;

/\*\*

\* The path to the file the stream is writing to as specified in the first

\* argument to {@link createWriteStream}. If `path` is passed as a string, then`writeStream.path` will be a string. If `path` is passed as a `Buffer`, then`writeStream.path` will be a

\* `Buffer`.

\* @since v0.1.93

\*/

path: string | Buffer;

/\*\*

\* This property is `true` if the underlying file has not been opened yet,

\* i.e. before the `'ready'` event is emitted.

\* @since v11.2.0

\*/

pending: boolean;

/\*\*

\* events.EventEmitter

\* 1. open

\* 2. close

\* 3. ready

\*/

addListener(event: 'close', listener: () => void): this;

addListener(event: 'drain', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'finish', listener: () => void): this;

addListener(event: 'open', listener: (fd: number) => void): this;

addListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

addListener(event: 'ready', listener: () => void): this;

addListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

on(event: 'close', listener: () => void): this;

on(event: 'drain', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'finish', listener: () => void): this;

on(event: 'open', listener: (fd: number) => void): this;

on(event: 'pipe', listener: (src: stream.Readable) => void): this;

on(event: 'ready', listener: () => void): this;

on(event: 'unpipe', listener: (src: stream.Readable) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'drain', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'finish', listener: () => void): this;

once(event: 'open', listener: (fd: number) => void): this;

once(event: 'pipe', listener: (src: stream.Readable) => void): this;

once(event: 'ready', listener: () => void): this;

once(event: 'unpipe', listener: (src: stream.Readable) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'drain', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'finish', listener: () => void): this;

prependListener(event: 'open', listener: (fd: number) => void): this;

prependListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

prependListener(event: 'ready', listener: () => void): this;

prependListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'drain', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'finish', listener: () => void): this;

prependOnceListener(event: 'open', listener: (fd: number) => void): this;

prependOnceListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

prependOnceListener(event: 'ready', listener: () => void): this;

prependOnceListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

/\*\*

\* Asynchronously rename file at `oldPath` to the pathname provided

\* as `newPath`. In the case that `newPath` already exists, it will

\* be overwritten. If there is a directory at `newPath`, an error will

\* be raised instead. No arguments other than a possible exception are

\* given to the completion callback.

\*

\* See also: [`rename(2)`](http://man7.org/linux/man-pages/man2/rename.2.html).

\*

\* ```js

\* import { rename } from 'fs';

\*

\* rename('oldFile.txt', 'newFile.txt', (err) => {

\* if (err) throw err;

\* console.log('Rename complete!');

\* });

\* ```

\* @since v0.0.2

\*/

export function rename(oldPath: PathLike, newPath: PathLike, callback: NoParamCallback): void;

export namespace rename {

/\*\*

\* Asynchronous rename(2) - Change the name or location of a file or directory.

\* @param oldPath A path to a file. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\* @param newPath A path to a file. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\*/

function \_\_promisify\_\_(oldPath: PathLike, newPath: PathLike): Promise<void>;

}

/\*\*

\* Renames the file from `oldPath` to `newPath`. Returns `undefined`.

\*

\* See the POSIX [`rename(2)`](http://man7.org/linux/man-pages/man2/rename.2.html) documentation for more details.

\* @since v0.1.21

\*/

export function renameSync(oldPath: PathLike, newPath: PathLike): void;

/\*\*

\* Truncates the file. No arguments other than a possible exception are

\* given to the completion callback. A file descriptor can also be passed as the

\* first argument. In this case, `fs.ftruncate()` is called.

\*

\* ```js

\* import { truncate } from 'fs';

\* // Assuming that 'path/file.txt' is a regular file.

\* truncate('path/file.txt', (err) => {

\* if (err) throw err;

\* console.log('path/file.txt was truncated');

\* });

\* ```

\*

\* Passing a file descriptor is deprecated and may result in an error being thrown

\* in the future.

\*

\* See the POSIX [`truncate(2)`](http://man7.org/linux/man-pages/man2/truncate.2.html) documentation for more details.

\* @since v0.8.6

\* @param [len=0]

\*/

export function truncate(path: PathLike, len: number | undefined | null, callback: NoParamCallback): void;

/\*\*

\* Asynchronous truncate(2) - Truncate a file to a specified length.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export function truncate(path: PathLike, callback: NoParamCallback): void;

export namespace truncate {

/\*\*

\* Asynchronous truncate(2) - Truncate a file to a specified length.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param len If not specified, defaults to `0`.

\*/

function \_\_promisify\_\_(path: PathLike, len?: number | null): Promise<void>;

}

/\*\*

\* Truncates the file. Returns `undefined`. A file descriptor can also be

\* passed as the first argument. In this case, `fs.ftruncateSync()` is called.

\*

\* Passing a file descriptor is deprecated and may result in an error being thrown

\* in the future.

\* @since v0.8.6

\* @param [len=0]

\*/

export function truncateSync(path: PathLike, len?: number | null): void;

/\*\*

\* Truncates the file descriptor. No arguments other than a possible exception are

\* given to the completion callback.

\*

\* See the POSIX [`ftruncate(2)`](http://man7.org/linux/man-pages/man2/ftruncate.2.html) documentation for more detail.

\*

\* If the file referred to by the file descriptor was larger than `len` bytes, only

\* the first `len` bytes will be retained in the file.

\*

\* For example, the following program retains only the first four bytes of the

\* file:

\*

\* ```js

\* import { open, close, ftruncate } from 'fs';

\*

\* function closeFd(fd) {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\*

\* open('temp.txt', 'r+', (err, fd) => {

\* if (err) throw err;

\*

\* try {

\* ftruncate(fd, 4, (err) => {

\* closeFd(fd);

\* if (err) throw err;

\* });

\* } catch (err) {

\* closeFd(fd);

\* if (err) throw err;

\* }

\* });

\* ```

\*

\* If the file previously was shorter than `len` bytes, it is extended, and the

\* extended part is filled with null bytes (`'\0'`):

\*

\* If `len` is negative then `0` will be used.

\* @since v0.8.6

\* @param [len=0]

\*/

export function ftruncate(fd: number, len: number | undefined | null, callback: NoParamCallback): void;

/\*\*

\* Asynchronous ftruncate(2) - Truncate a file to a specified length.

\* @param fd A file descriptor.

\*/

export function ftruncate(fd: number, callback: NoParamCallback): void;

export namespace ftruncate {

/\*\*

\* Asynchronous ftruncate(2) - Truncate a file to a specified length.

\* @param fd A file descriptor.

\* @param len If not specified, defaults to `0`.

\*/

function \_\_promisify\_\_(fd: number, len?: number | null): Promise<void>;

}

/\*\*

\* Truncates the file descriptor. Returns `undefined`.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link ftruncate}.

\* @since v0.8.6

\* @param [len=0]

\*/

export function ftruncateSync(fd: number, len?: number | null): void;

/\*\*

\* Asynchronously changes owner and group of a file. No arguments other than a

\* possible exception are given to the completion callback.

\*

\* See the POSIX [`chown(2)`](http://man7.org/linux/man-pages/man2/chown.2.html) documentation for more detail.

\* @since v0.1.97

\*/

export function chown(path: PathLike, uid: number, gid: number, callback: NoParamCallback): void;

export namespace chown {

/\*\*

\* Asynchronous chown(2) - Change ownership of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(path: PathLike, uid: number, gid: number): Promise<void>;

}

/\*\*

\* Synchronously changes owner and group of a file. Returns `undefined`.

\* This is the synchronous version of {@link chown}.

\*

\* See the POSIX [`chown(2)`](http://man7.org/linux/man-pages/man2/chown.2.html) documentation for more detail.

\* @since v0.1.97

\*/

export function chownSync(path: PathLike, uid: number, gid: number): void;

/\*\*

\* Sets the owner of the file. No arguments other than a possible exception are

\* given to the completion callback.

\*

\* See the POSIX [`fchown(2)`](http://man7.org/linux/man-pages/man2/fchown.2.html) documentation for more detail.

\* @since v0.4.7

\*/

export function fchown(fd: number, uid: number, gid: number, callback: NoParamCallback): void;

export namespace fchown {

/\*\*

\* Asynchronous fchown(2) - Change ownership of a file.

\* @param fd A file descriptor.

\*/

function \_\_promisify\_\_(fd: number, uid: number, gid: number): Promise<void>;

}

/\*\*

\* Sets the owner of the file. Returns `undefined`.

\*

\* See the POSIX [`fchown(2)`](http://man7.org/linux/man-pages/man2/fchown.2.html) documentation for more detail.

\* @since v0.4.7

\* @param uid The file's new owner's user id.

\* @param gid The file's new group's group id.

\*/

export function fchownSync(fd: number, uid: number, gid: number): void;

/\*\*

\* Set the owner of the symbolic link. No arguments other than a possible

\* exception are given to the completion callback.

\*

\* See the POSIX [`lchown(2)`](http://man7.org/linux/man-pages/man2/lchown.2.html) documentation for more detail.

\*/

export function lchown(path: PathLike, uid: number, gid: number, callback: NoParamCallback): void;

export namespace lchown {

/\*\*

\* Asynchronous lchown(2) - Change ownership of a file. Does not dereference symbolic links.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(path: PathLike, uid: number, gid: number): Promise<void>;

}

/\*\*

\* Set the owner for the path. Returns `undefined`.

\*

\* See the POSIX [`lchown(2)`](http://man7.org/linux/man-pages/man2/lchown.2.html) documentation for more details.

\* @param uid The file's new owner's user id.

\* @param gid The file's new group's group id.

\*/

export function lchownSync(path: PathLike, uid: number, gid: number): void;

/\*\*

\* Changes the access and modification times of a file in the same way as {@link utimes}, with the difference that if the path refers to a symbolic

\* link, then the link is not dereferenced: instead, the timestamps of the

\* symbolic link itself are changed.

\*

\* No arguments other than a possible exception are given to the completion

\* callback.

\* @since v14.5.0, v12.19.0

\*/

export function lutimes(path: PathLike, atime: TimeLike, mtime: TimeLike, callback: NoParamCallback): void;

export namespace lutimes {

/\*\*

\* Changes the access and modification times of a file in the same way as `fsPromises.utimes()`,

\* with the difference that if the path refers to a symbolic link, then the link is not

\* dereferenced: instead, the timestamps of the symbolic link itself are changed.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param atime The last access time. If a string is provided, it will be coerced to number.

\* @param mtime The last modified time. If a string is provided, it will be coerced to number.

\*/

function \_\_promisify\_\_(path: PathLike, atime: TimeLike, mtime: TimeLike): Promise<void>;

}

/\*\*

\* Change the file system timestamps of the symbolic link referenced by `path`.

\* Returns `undefined`, or throws an exception when parameters are incorrect or

\* the operation fails. This is the synchronous version of {@link lutimes}.

\* @since v14.5.0, v12.19.0

\*/

export function lutimesSync(path: PathLike, atime: TimeLike, mtime: TimeLike): void;

/\*\*

\* Asynchronously changes the permissions of a file. No arguments other than a

\* possible exception are given to the completion callback.

\*

\* See the POSIX [`chmod(2)`](http://man7.org/linux/man-pages/man2/chmod.2.html) documentation for more detail.

\*

\* ```js

\* import { chmod } from 'fs';

\*

\* chmod('my\_file.txt', 0o775, (err) => {

\* if (err) throw err;

\* console.log('The permissions for file "my\_file.txt" have been changed!');

\* });

\* ```

\* @since v0.1.30

\*/

export function chmod(path: PathLike, mode: Mode, callback: NoParamCallback): void;

export namespace chmod {

/\*\*

\* Asynchronous chmod(2) - Change permissions of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param mode A file mode. If a string is passed, it is parsed as an octal integer.

\*/

function \_\_promisify\_\_(path: PathLike, mode: Mode): Promise<void>;

}

/\*\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link chmod}.

\*

\* See the POSIX [`chmod(2)`](http://man7.org/linux/man-pages/man2/chmod.2.html) documentation for more detail.

\* @since v0.6.7

\*/

export function chmodSync(path: PathLike, mode: Mode): void;

/\*\*

\* Sets the permissions on the file. No arguments other than a possible exception

\* are given to the completion callback.

\*

\* See the POSIX [`fchmod(2)`](http://man7.org/linux/man-pages/man2/fchmod.2.html) documentation for more detail.

\* @since v0.4.7

\*/

export function fchmod(fd: number, mode: Mode, callback: NoParamCallback): void;

export namespace fchmod {

/\*\*

\* Asynchronous fchmod(2) - Change permissions of a file.

\* @param fd A file descriptor.

\* @param mode A file mode. If a string is passed, it is parsed as an octal integer.

\*/

function \_\_promisify\_\_(fd: number, mode: Mode): Promise<void>;

}

/\*\*

\* Sets the permissions on the file. Returns `undefined`.

\*

\* See the POSIX [`fchmod(2)`](http://man7.org/linux/man-pages/man2/fchmod.2.html) documentation for more detail.

\* @since v0.4.7

\*/

export function fchmodSync(fd: number, mode: Mode): void;

/\*\*

\* Changes the permissions on a symbolic link. No arguments other than a possible

\* exception are given to the completion callback.

\*

\* This method is only implemented on macOS.

\*

\* See the POSIX [`lchmod(2)`](https://www.freebsd.org/cgi/man.cgi?query=lchmod&sektion=2) documentation for more detail.

\* @deprecated Since v0.4.7

\*/

export function lchmod(path: PathLike, mode: Mode, callback: NoParamCallback): void;

/\*\* @deprecated \*/

export namespace lchmod {

/\*\*

\* Asynchronous lchmod(2) - Change permissions of a file. Does not dereference symbolic links.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param mode A file mode. If a string is passed, it is parsed as an octal integer.

\*/

function \_\_promisify\_\_(path: PathLike, mode: Mode): Promise<void>;

}

/\*\*

\* Changes the permissions on a symbolic link. Returns `undefined`.

\*

\* This method is only implemented on macOS.

\*

\* See the POSIX [`lchmod(2)`](https://www.freebsd.org/cgi/man.cgi?query=lchmod&sektion=2) documentation for more detail.

\* @deprecated Since v0.4.7

\*/

export function lchmodSync(path: PathLike, mode: Mode): void;

/\*\*

\* Asynchronous [`stat(2)`](http://man7.org/linux/man-pages/man2/stat.2.html). The callback gets two arguments `(err, stats)` where`stats` is an `fs.Stats` object.

\*

\* In case of an error, the `err.code` will be one of `Common System Errors`.

\*

\* Using `fs.stat()` to check for the existence of a file before calling`fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended.

\* Instead, user code should open/read/write the file directly and handle the

\* error raised if the file is not available.

\*

\* To check if a file exists without manipulating it afterwards, {@link access} is recommended.

\*

\* For example, given the following directory structure:

\*

\* ```text

\* - txtDir

\* -- file.txt

\* - app.js

\* ```

\*

\* The next program will check for the stats of the given paths:

\*

\* ```js

\* import { stat } from 'fs';

\*

\* const pathsToCheck = ['./txtDir', './txtDir/file.txt'];

\*

\* for (let i = 0; i < pathsToCheck.length; i++) {

\* stat(pathsToCheck[i], (err, stats) => {

\* console.log(stats.isDirectory());

\* console.log(stats);

\* });

\* }

\* ```

\*

\* The resulting output will resemble:

\*

\* ```console

\* true

\* Stats {

\* dev: 16777220,

\* mode: 16877,

\* nlink: 3,

\* uid: 501,

\* gid: 20,

\* rdev: 0,

\* blksize: 4096,

\* ino: 14214262,

\* size: 96,

\* blocks: 0,

\* atimeMs: 1561174653071.963,

\* mtimeMs: 1561174614583.3518,

\* ctimeMs: 1561174626623.5366,

\* birthtimeMs: 1561174126937.2893,

\* atime: 2019-06-22T03:37:33.072Z,

\* mtime: 2019-06-22T03:36:54.583Z,

\* ctime: 2019-06-22T03:37:06.624Z,

\* birthtime: 2019-06-22T03:28:46.937Z

\* }

\* false

\* Stats {

\* dev: 16777220,

\* mode: 33188,

\* nlink: 1,

\* uid: 501,

\* gid: 20,

\* rdev: 0,

\* blksize: 4096,

\* ino: 14214074,

\* size: 8,

\* blocks: 8,

\* atimeMs: 1561174616618.8555,

\* mtimeMs: 1561174614584,

\* ctimeMs: 1561174614583.8145,

\* birthtimeMs: 1561174007710.7478,

\* atime: 2019-06-22T03:36:56.619Z,

\* mtime: 2019-06-22T03:36:54.584Z,

\* ctime: 2019-06-22T03:36:54.584Z,

\* birthtime: 2019-06-22T03:26:47.711Z

\* }

\* ```

\* @since v0.0.2

\*/

export function stat(path: PathLike, callback: (err: NodeJS.ErrnoException | null, stats: Stats) => void): void;

export function stat(

path: PathLike,

options:

| (StatOptions & {

bigint?: false | undefined;

})

| undefined,

callback: (err: NodeJS.ErrnoException | null, stats: Stats) => void

): void;

export function stat(

path: PathLike,

options: StatOptions & {

bigint: true;

},

callback: (err: NodeJS.ErrnoException | null, stats: BigIntStats) => void

): void;

export function stat(path: PathLike, options: StatOptions | undefined, callback: (err: NodeJS.ErrnoException | null, stats: Stats | BigIntStats) => void): void;

export namespace stat {

/\*\*

\* Asynchronous stat(2) - Get file status.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(

path: PathLike,

options?: StatOptions & {

bigint?: false | undefined;

}

): Promise<Stats>;

function \_\_promisify\_\_(

path: PathLike,

options: StatOptions & {

bigint: true;

}

): Promise<BigIntStats>;

function \_\_promisify\_\_(path: PathLike, options?: StatOptions): Promise<Stats | BigIntStats>;

}

export interface StatSyncFn extends Function {

(path: PathLike, options?: undefined): Stats;

(

path: PathLike,

options?: StatSyncOptions & {

bigint?: false | undefined;

throwIfNoEntry: false;

}

): Stats | undefined;

(

path: PathLike,

options: StatSyncOptions & {

bigint: true;

throwIfNoEntry: false;

}

): BigIntStats | undefined;

(

path: PathLike,

options?: StatSyncOptions & {

bigint?: false | undefined;

}

): Stats;

(

path: PathLike,

options: StatSyncOptions & {

bigint: true;

}

): BigIntStats;

(

path: PathLike,

options: StatSyncOptions & {

bigint: boolean;

throwIfNoEntry?: false | undefined;

}

): Stats | BigIntStats;

(path: PathLike, options?: StatSyncOptions): Stats | BigIntStats | undefined;

}

/\*\*

\* Synchronous stat(2) - Get file status.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export const statSync: StatSyncFn;

/\*\*

\* Invokes the callback with the `fs.Stats` for the file descriptor.

\*

\* See the POSIX [`fstat(2)`](http://man7.org/linux/man-pages/man2/fstat.2.html) documentation for more detail.

\* @since v0.1.95

\*/

export function fstat(fd: number, callback: (err: NodeJS.ErrnoException | null, stats: Stats) => void): void;

export function fstat(

fd: number,

options:

| (StatOptions & {

bigint?: false | undefined;

})

| undefined,

callback: (err: NodeJS.ErrnoException | null, stats: Stats) => void

): void;

export function fstat(

fd: number,

options: StatOptions & {

bigint: true;

},

callback: (err: NodeJS.ErrnoException | null, stats: BigIntStats) => void

): void;

export function fstat(fd: number, options: StatOptions | undefined, callback: (err: NodeJS.ErrnoException | null, stats: Stats | BigIntStats) => void): void;

export namespace fstat {

/\*\*

\* Asynchronous fstat(2) - Get file status.

\* @param fd A file descriptor.

\*/

function \_\_promisify\_\_(

fd: number,

options?: StatOptions & {

bigint?: false | undefined;

}

): Promise<Stats>;

function \_\_promisify\_\_(

fd: number,

options: StatOptions & {

bigint: true;

}

): Promise<BigIntStats>;

function \_\_promisify\_\_(fd: number, options?: StatOptions): Promise<Stats | BigIntStats>;

}

/\*\*

\* Retrieves the `fs.Stats` for the file descriptor.

\*

\* See the POSIX [`fstat(2)`](http://man7.org/linux/man-pages/man2/fstat.2.html) documentation for more detail.

\* @since v0.1.95

\*/

export function fstatSync(

fd: number,

options?: StatOptions & {

bigint?: false | undefined;

}

): Stats;

export function fstatSync(

fd: number,

options: StatOptions & {

bigint: true;

}

): BigIntStats;

export function fstatSync(fd: number, options?: StatOptions): Stats | BigIntStats;

/\*\*

\* Retrieves the `fs.Stats` for the symbolic link referred to by the path.

\* The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `lstat()` is identical to `stat()`, except that if `path` is a symbolic

\* link, then the link itself is stat-ed, not the file that it refers to.

\*

\* See the POSIX [`lstat(2)`](http://man7.org/linux/man-pages/man2/lstat.2.html) documentation for more details.

\* @since v0.1.30

\*/

export function lstat(path: PathLike, callback: (err: NodeJS.ErrnoException | null, stats: Stats) => void): void;

export function lstat(

path: PathLike,

options:

| (StatOptions & {

bigint?: false | undefined;

})

| undefined,

callback: (err: NodeJS.ErrnoException | null, stats: Stats) => void

): void;

export function lstat(

path: PathLike,

options: StatOptions & {

bigint: true;

},

callback: (err: NodeJS.ErrnoException | null, stats: BigIntStats) => void

): void;

export function lstat(path: PathLike, options: StatOptions | undefined, callback: (err: NodeJS.ErrnoException | null, stats: Stats | BigIntStats) => void): void;

export namespace lstat {

/\*\*

\* Asynchronous lstat(2) - Get file status. Does not dereference symbolic links.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(

path: PathLike,

options?: StatOptions & {

bigint?: false | undefined;

}

): Promise<Stats>;

function \_\_promisify\_\_(

path: PathLike,

options: StatOptions & {

bigint: true;

}

): Promise<BigIntStats>;

function \_\_promisify\_\_(path: PathLike, options?: StatOptions): Promise<Stats | BigIntStats>;

}

/\*\*

\* Synchronous lstat(2) - Get file status. Does not dereference symbolic links.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export const lstatSync: StatSyncFn;

/\*\*

\* Creates a new link from the `existingPath` to the `newPath`. See the POSIX [`link(2)`](http://man7.org/linux/man-pages/man2/link.2.html) documentation for more detail. No arguments other than

\* a possible

\* exception are given to the completion callback.

\* @since v0.1.31

\*/

export function link(existingPath: PathLike, newPath: PathLike, callback: NoParamCallback): void;

export namespace link {

/\*\*

\* Asynchronous link(2) - Create a new link (also known as a hard link) to an existing file.

\* @param existingPath A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param newPath A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(existingPath: PathLike, newPath: PathLike): Promise<void>;

}

/\*\*

\* Creates a new link from the `existingPath` to the `newPath`. See the POSIX [`link(2)`](http://man7.org/linux/man-pages/man2/link.2.html) documentation for more detail. Returns `undefined`.

\* @since v0.1.31

\*/

export function linkSync(existingPath: PathLike, newPath: PathLike): void;

/\*\*

\* Creates the link called `path` pointing to `target`. No arguments other than a

\* possible exception are given to the completion callback.

\*

\* See the POSIX [`symlink(2)`](http://man7.org/linux/man-pages/man2/symlink.2.html) documentation for more details.

\*

\* The `type` argument is only available on Windows and ignored on other platforms.

\* It can be set to `'dir'`, `'file'`, or `'junction'`. If the `type` argument is

\* not set, Node.js will autodetect `target` type and use `'file'` or `'dir'`. If

\* the `target` does not exist, `'file'` will be used. Windows junction points

\* require the destination path to be absolute. When using `'junction'`, the`target` argument will automatically be normalized to absolute path.

\*

\* Relative targets are relative to the link’s parent directory.

\*

\* ```js

\* import { symlink } from 'fs';

\*

\* symlink('./mew', './example/mewtwo', callback);

\* ```

\*

\* The above example creates a symbolic link `mewtwo` in the `example` which points

\* to `mew` in the same directory:

\*

\* ```bash

\* $ tree example/

\* example/

\* ├── mew

\* └── mewtwo -> ./mew

\* ```

\* @since v0.1.31

\*/

export function symlink(target: PathLike, path: PathLike, type: symlink.Type | undefined | null, callback: NoParamCallback): void;

/\*\*

\* Asynchronous symlink(2) - Create a new symbolic link to an existing file.

\* @param target A path to an existing file. If a URL is provided, it must use the `file:` protocol.

\* @param path A path to the new symlink. If a URL is provided, it must use the `file:` protocol.

\*/

export function symlink(target: PathLike, path: PathLike, callback: NoParamCallback): void;

export namespace symlink {

/\*\*

\* Asynchronous symlink(2) - Create a new symbolic link to an existing file.

\* @param target A path to an existing file. If a URL is provided, it must use the `file:` protocol.

\* @param path A path to the new symlink. If a URL is provided, it must use the `file:` protocol.

\* @param type May be set to `'dir'`, `'file'`, or `'junction'` (default is `'file'`) and is only available on Windows (ignored on other platforms).

\* When using `'junction'`, the `target` argument will automatically be normalized to an absolute path.

\*/

function \_\_promisify\_\_(target: PathLike, path: PathLike, type?: string | null): Promise<void>;

type Type = 'dir' | 'file' | 'junction';

}

/\*\*

\* Returns `undefined`.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link symlink}.

\* @since v0.1.31

\*/

export function symlinkSync(target: PathLike, path: PathLike, type?: symlink.Type | null): void;

/\*\*

\* Reads the contents of the symbolic link referred to by `path`. The callback gets

\* two arguments `(err, linkString)`.

\*

\* See the POSIX [`readlink(2)`](http://man7.org/linux/man-pages/man2/readlink.2.html) documentation for more details.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use for

\* the link path passed to the callback. If the `encoding` is set to `'buffer'`,

\* the link path returned will be passed as a `Buffer` object.

\* @since v0.1.31

\*/

export function readlink(path: PathLike, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, linkString: string) => void): void;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readlink(path: PathLike, options: BufferEncodingOption, callback: (err: NodeJS.ErrnoException | null, linkString: Buffer) => void): void;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readlink(path: PathLike, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, linkString: string | Buffer) => void): void;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export function readlink(path: PathLike, callback: (err: NodeJS.ErrnoException | null, linkString: string) => void): void;

export namespace readlink {

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(path: PathLike, options?: EncodingOption): Promise<string>;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(path: PathLike, options: BufferEncodingOption): Promise<Buffer>;

/\*\*

\* Asynchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(path: PathLike, options?: EncodingOption): Promise<string | Buffer>;

}

/\*\*

\* Returns the symbolic link's string value.

\*

\* See the POSIX [`readlink(2)`](http://man7.org/linux/man-pages/man2/readlink.2.html) documentation for more details.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use for

\* the link path returned. If the `encoding` is set to `'buffer'`,

\* the link path returned will be passed as a `Buffer` object.

\* @since v0.1.31

\*/

export function readlinkSync(path: PathLike, options?: EncodingOption): string;

/\*\*

\* Synchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readlinkSync(path: PathLike, options: BufferEncodingOption): Buffer;

/\*\*

\* Synchronous readlink(2) - read value of a symbolic link.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readlinkSync(path: PathLike, options?: EncodingOption): string | Buffer;

/\*\*

\* Asynchronously computes the canonical pathname by resolving `.`, `..` and

\* symbolic links.

\*

\* A canonical pathname is not necessarily unique. Hard links and bind mounts can

\* expose a file system entity through many pathnames.

\*

\* This function behaves like [`realpath(3)`](http://man7.org/linux/man-pages/man3/realpath.3.html), with some exceptions:

\*

\* 1. No case conversion is performed on case-insensitive file systems.

\* 2. The maximum number of symbolic links is platform-independent and generally

\* (much) higher than what the native [`realpath(3)`](http://man7.org/linux/man-pages/man3/realpath.3.html) implementation supports.

\*

\* The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd`to resolve relative paths.

\*

\* Only paths that can be converted to UTF8 strings are supported.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use for

\* the path passed to the callback. If the `encoding` is set to `'buffer'`,

\* the path returned will be passed as a `Buffer` object.

\*

\* If `path` resolves to a socket or a pipe, the function will return a system

\* dependent name for that object.

\* @since v0.1.31

\*/

export function realpath(path: PathLike, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, resolvedPath: string) => void): void;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function realpath(path: PathLike, options: BufferEncodingOption, callback: (err: NodeJS.ErrnoException | null, resolvedPath: Buffer) => void): void;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function realpath(path: PathLike, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, resolvedPath: string | Buffer) => void): void;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export function realpath(path: PathLike, callback: (err: NodeJS.ErrnoException | null, resolvedPath: string) => void): void;

export namespace realpath {

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(path: PathLike, options?: EncodingOption): Promise<string>;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(path: PathLike, options: BufferEncodingOption): Promise<Buffer>;

/\*\*

\* Asynchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(path: PathLike, options?: EncodingOption): Promise<string | Buffer>;

/\*\*

\* Asynchronous [`realpath(3)`](http://man7.org/linux/man-pages/man3/realpath.3.html).

\*

\* The `callback` gets two arguments `(err, resolvedPath)`.

\*

\* Only paths that can be converted to UTF8 strings are supported.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use for

\* the path passed to the callback. If the `encoding` is set to `'buffer'`,

\* the path returned will be passed as a `Buffer` object.

\*

\* On Linux, when Node.js is linked against musl libc, the procfs file system must

\* be mounted on `/proc` in order for this function to work. Glibc does not have

\* this restriction.

\* @since v9.2.0

\*/

function native(path: PathLike, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, resolvedPath: string) => void): void;

function native(path: PathLike, options: BufferEncodingOption, callback: (err: NodeJS.ErrnoException | null, resolvedPath: Buffer) => void): void;

function native(path: PathLike, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, resolvedPath: string | Buffer) => void): void;

function native(path: PathLike, callback: (err: NodeJS.ErrnoException | null, resolvedPath: string) => void): void;

}

/\*\*

\* Returns the resolved pathname.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link realpath}.

\* @since v0.1.31

\*/

export function realpathSync(path: PathLike, options?: EncodingOption): string;

/\*\*

\* Synchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function realpathSync(path: PathLike, options: BufferEncodingOption): Buffer;

/\*\*

\* Synchronous realpath(3) - return the canonicalized absolute pathname.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function realpathSync(path: PathLike, options?: EncodingOption): string | Buffer;

export namespace realpathSync {

function native(path: PathLike, options?: EncodingOption): string;

function native(path: PathLike, options: BufferEncodingOption): Buffer;

function native(path: PathLike, options?: EncodingOption): string | Buffer;

}

/\*\*

\* Asynchronously removes a file or symbolic link. No arguments other than a

\* possible exception are given to the completion callback.

\*

\* ```js

\* import { unlink } from 'fs';

\* // Assuming that 'path/file.txt' is a regular file.

\* unlink('path/file.txt', (err) => {

\* if (err) throw err;

\* console.log('path/file.txt was deleted');

\* });

\* ```

\*

\* `fs.unlink()` will not work on a directory, empty or otherwise. To remove a

\* directory, use {@link rmdir}.

\*

\* See the POSIX [`unlink(2)`](http://man7.org/linux/man-pages/man2/unlink.2.html) documentation for more details.

\* @since v0.0.2

\*/

export function unlink(path: PathLike, callback: NoParamCallback): void;

export namespace unlink {

/\*\*

\* Asynchronous unlink(2) - delete a name and possibly the file it refers to.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(path: PathLike): Promise<void>;

}

/\*\*

\* Synchronous [`unlink(2)`](http://man7.org/linux/man-pages/man2/unlink.2.html). Returns `undefined`.

\* @since v0.1.21

\*/

export function unlinkSync(path: PathLike): void;

export interface RmDirOptions {

/\*\*

\* If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or

\* `EPERM` error is encountered, Node.js will retry the operation with a linear

\* backoff wait of `retryDelay` ms longer on each try. This option represents the

\* number of retries. This option is ignored if the `recursive` option is not

\* `true`.

\* @default 0

\*/

maxRetries?: number | undefined;

/\*\*

\* @deprecated since v14.14.0 In future versions of Node.js and will trigger a warning

\* `fs.rmdir(path, { recursive: true })` will throw if `path` does not exist or is a file.

\* Use `fs.rm(path, { recursive: true, force: true })` instead.

\*

\* If `true`, perform a recursive directory removal. In

\* recursive mode soperations are retried on failure.

\* @default false

\*/

recursive?: boolean | undefined;

/\*\*

\* The amount of time in milliseconds to wait between retries.

\* This option is ignored if the `recursive` option is not `true`.

\* @default 100

\*/

retryDelay?: number | undefined;

}

/\*\*

\* Asynchronous [`rmdir(2)`](http://man7.org/linux/man-pages/man2/rmdir.2.html). No arguments other than a possible exception are given

\* to the completion callback.

\*

\* Using `fs.rmdir()` on a file (not a directory) results in an `ENOENT` error on

\* Windows and an `ENOTDIR` error on POSIX.

\*

\* To get a behavior similar to the `rm -rf` Unix command, use {@link rm} with options `{ recursive: true, force: true }`.

\* @since v0.0.2

\*/

export function rmdir(path: PathLike, callback: NoParamCallback): void;

export function rmdir(path: PathLike, options: RmDirOptions, callback: NoParamCallback): void;

export namespace rmdir {

/\*\*

\* Asynchronous rmdir(2) - delete a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

function \_\_promisify\_\_(path: PathLike, options?: RmDirOptions): Promise<void>;

}

/\*\*

\* Synchronous [`rmdir(2)`](http://man7.org/linux/man-pages/man2/rmdir.2.html). Returns `undefined`.

\*

\* Using `fs.rmdirSync()` on a file (not a directory) results in an `ENOENT` error

\* on Windows and an `ENOTDIR` error on POSIX.

\*

\* To get a behavior similar to the `rm -rf` Unix command, use {@link rmSync} with options `{ recursive: true, force: true }`.

\* @since v0.1.21

\*/

export function rmdirSync(path: PathLike, options?: RmDirOptions): void;

export interface RmOptions {

/\*\*

\* When `true`, exceptions will be ignored if `path` does not exist.

\* @default false

\*/

force?: boolean | undefined;

/\*\*

\* If an `EBUSY`, `EMFILE`, `ENFILE`, `ENOTEMPTY`, or

\* `EPERM` error is encountered, Node.js will retry the operation with a linear

\* backoff wait of `retryDelay` ms longer on each try. This option represents the

\* number of retries. This option is ignored if the `recursive` option is not

\* `true`.

\* @default 0

\*/

maxRetries?: number | undefined;

/\*\*

\* If `true`, perform a recursive directory removal. In

\* recursive mode, operations are retried on failure.

\* @default false

\*/

recursive?: boolean | undefined;

/\*\*

\* The amount of time in milliseconds to wait between retries.

\* This option is ignored if the `recursive` option is not `true`.

\* @default 100

\*/

retryDelay?: number | undefined;

}

/\*\*

\* Asynchronously removes files and directories (modeled on the standard POSIX `rm`utility). No arguments other than a possible exception are given to the

\* completion callback.

\* @since v14.14.0

\*/

export function rm(path: PathLike, callback: NoParamCallback): void;

export function rm(path: PathLike, options: RmOptions, callback: NoParamCallback): void;

export namespace rm {

/\*\*

\* Asynchronously removes files and directories (modeled on the standard POSIX `rm` utility).

\*/

function \_\_promisify\_\_(path: PathLike, options?: RmOptions): Promise<void>;

}

/\*\*

\* Synchronously removes files and directories (modeled on the standard POSIX `rm`utility). Returns `undefined`.

\* @since v14.14.0

\*/

export function rmSync(path: PathLike, options?: RmOptions): void;

export interface MakeDirectoryOptions {

/\*\*

\* Indicates whether parent folders should be created.

\* If a folder was created, the path to the first created folder will be returned.

\* @default false

\*/

recursive?: boolean | undefined;

/\*\*

\* A file mode. If a string is passed, it is parsed as an octal integer. If not specified

\* @default 0o777

\*/

mode?: Mode | undefined;

}

/\*\*

\* Asynchronously creates a directory.

\*

\* The callback is given a possible exception and, if `recursive` is `true`, the

\* first directory path created, `(err[, path])`.`path` can still be `undefined` when `recursive` is `true`, if no directory was

\* created.

\*

\* The optional `options` argument can be an integer specifying `mode` (permission

\* and sticky bits), or an object with a `mode` property and a `recursive`property indicating whether parent directories should be created. Calling`fs.mkdir()` when `path` is a directory that

\* exists results in an error only

\* when `recursive` is false.

\*

\* ```js

\* import { mkdir } from 'fs';

\*

\* // Creates /tmp/a/apple, regardless of whether `/tmp` and /tmp/a exist.

\* mkdir('/tmp/a/apple', { recursive: true }, (err) => {

\* if (err) throw err;

\* });

\* ```

\*

\* On Windows, using `fs.mkdir()` on the root directory even with recursion will

\* result in an error:

\*

\* ```js

\* import { mkdir } from 'fs';

\*

\* mkdir('/', { recursive: true }, (err) => {

\* // => [Error: EPERM: operation not permitted, mkdir 'C:\']

\* });

\* ```

\*

\* See the POSIX [`mkdir(2)`](http://man7.org/linux/man-pages/man2/mkdir.2.html) documentation for more details.

\* @since v0.1.8

\*/

export function mkdir(

path: PathLike,

options: MakeDirectoryOptions & {

recursive: true;

},

callback: (err: NodeJS.ErrnoException | null, path?: string) => void

): void;

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

export function mkdir(

path: PathLike,

options:

| Mode

| (MakeDirectoryOptions & {

recursive?: false | undefined;

})

| null

| undefined,

callback: NoParamCallback

): void;

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

export function mkdir(path: PathLike, options: Mode | MakeDirectoryOptions | null | undefined, callback: (err: NodeJS.ErrnoException | null, path?: string) => void): void;

/\*\*

\* Asynchronous mkdir(2) - create a directory with a mode of `0o777`.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export function mkdir(path: PathLike, callback: NoParamCallback): void;

export namespace mkdir {

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

function \_\_promisify\_\_(

path: PathLike,

options: MakeDirectoryOptions & {

recursive: true;

}

): Promise<string | undefined>;

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

function \_\_promisify\_\_(

path: PathLike,

options?:

| Mode

| (MakeDirectoryOptions & {

recursive?: false | undefined;

})

| null

): Promise<void>;

/\*\*

\* Asynchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

function \_\_promisify\_\_(path: PathLike, options?: Mode | MakeDirectoryOptions | null): Promise<string | undefined>;

}

/\*\*

\* Synchronously creates a directory. Returns `undefined`, or if `recursive` is`true`, the first directory path created.

\* This is the synchronous version of {@link mkdir}.

\*

\* See the POSIX [`mkdir(2)`](http://man7.org/linux/man-pages/man2/mkdir.2.html) documentation for more details.

\* @since v0.1.21

\*/

export function mkdirSync(

path: PathLike,

options: MakeDirectoryOptions & {

recursive: true;

}

): string | undefined;

/\*\*

\* Synchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

export function mkdirSync(

path: PathLike,

options?:

| Mode

| (MakeDirectoryOptions & {

recursive?: false | undefined;

})

| null

): void;

/\*\*

\* Synchronous mkdir(2) - create a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the file mode, or an object optionally specifying the file mode and whether parent folders

\* should be created. If a string is passed, it is parsed as an octal integer. If not specified, defaults to `0o777`.

\*/

export function mkdirSync(path: PathLike, options?: Mode | MakeDirectoryOptions | null): string | undefined;

/\*\*

\* Creates a unique temporary directory.

\*

\* Generates six random characters to be appended behind a required`prefix` to create a unique temporary directory. Due to platform

\* inconsistencies, avoid trailing `X` characters in `prefix`. Some platforms,

\* notably the BSDs, can return more than six random characters, and replace

\* trailing `X` characters in `prefix` with random characters.

\*

\* The created directory path is passed as a string to the callback's second

\* parameter.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use.

\*

\* ```js

\* import { mkdtemp } from 'fs';

\*

\* mkdtemp(path.join(os.tmpdir(), 'foo-'), (err, directory) => {

\* if (err) throw err;

\* console.log(directory);

\* // Prints: /tmp/foo-itXde2 or C:\Users\...\AppData\Local\Temp\foo-itXde2

\* });

\* ```

\*

\* The `fs.mkdtemp()` method will append the six randomly selected characters

\* directly to the `prefix` string. For instance, given a directory `/tmp`, if the

\* intention is to create a temporary directory \_within\_`/tmp`, the `prefix`must end with a trailing platform-specific path separator

\* (`require('path').sep`).

\*

\* ```js

\* import { tmpdir } from 'os';

\* import { mkdtemp } from 'fs';

\*

\* // The parent directory for the new temporary directory

\* const tmpDir = tmpdir();

\*

\* // This method is \*INCORRECT\*:

\* mkdtemp(tmpDir, (err, directory) => {

\* if (err) throw err;

\* console.log(directory);

\* // Will print something similar to `/tmpabc123`.

\* // A new temporary directory is created at the file system root

\* // rather than \*within\* the /tmp directory.

\* });

\*

\* // This method is \*CORRECT\*:

\* import { sep } from 'path';

\* mkdtemp(`${tmpDir}${sep}`, (err, directory) => {

\* if (err) throw err;

\* console.log(directory);

\* // Will print something similar to `/tmp/abc123`.

\* // A new temporary directory is created within

\* // the /tmp directory.

\* });

\* ```

\* @since v5.10.0

\*/

export function mkdtemp(prefix: string, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, folder: string) => void): void;

/\*\*

\* Asynchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function mkdtemp(

prefix: string,

options:

| 'buffer'

| {

encoding: 'buffer';

},

callback: (err: NodeJS.ErrnoException | null, folder: Buffer) => void

): void;

/\*\*

\* Asynchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function mkdtemp(prefix: string, options: EncodingOption, callback: (err: NodeJS.ErrnoException | null, folder: string | Buffer) => void): void;

/\*\*

\* Asynchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\*/

export function mkdtemp(prefix: string, callback: (err: NodeJS.ErrnoException | null, folder: string) => void): void;

export namespace mkdtemp {

/\*\*

\* Asynchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(prefix: string, options?: EncodingOption): Promise<string>;

/\*\*

\* Asynchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(prefix: string, options: BufferEncodingOption): Promise<Buffer>;

/\*\*

\* Asynchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(prefix: string, options?: EncodingOption): Promise<string | Buffer>;

}

/\*\*

\* Returns the created directory path.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link mkdtemp}.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use.

\* @since v5.10.0

\*/

export function mkdtempSync(prefix: string, options?: EncodingOption): string;

/\*\*

\* Synchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function mkdtempSync(prefix: string, options: BufferEncodingOption): Buffer;

/\*\*

\* Synchronously creates a unique temporary directory.

\* Generates six random characters to be appended behind a required prefix to create a unique temporary directory.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function mkdtempSync(prefix: string, options?: EncodingOption): string | Buffer;

/\*\*

\* Reads the contents of a directory. The callback gets two arguments `(err, files)`where `files` is an array of the names of the files in the directory excluding`'.'` and `'..'`.

\*

\* See the POSIX [`readdir(3)`](http://man7.org/linux/man-pages/man3/readdir.3.html) documentation for more details.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use for

\* the filenames passed to the callback. If the `encoding` is set to `'buffer'`,

\* the filenames returned will be passed as `Buffer` objects.

\*

\* If `options.withFileTypes` is set to `true`, the `files` array will contain `fs.Dirent` objects.

\* @since v0.1.8

\*/

export function readdir(

path: PathLike,

options:

| {

encoding: BufferEncoding | null;

withFileTypes?: false | undefined;

}

| BufferEncoding

| undefined

| null,

callback: (err: NodeJS.ErrnoException | null, files: string[]) => void

): void;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readdir(

path: PathLike,

options:

| {

encoding: 'buffer';

withFileTypes?: false | undefined;

}

| 'buffer',

callback: (err: NodeJS.ErrnoException | null, files: Buffer[]) => void

): void;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readdir(

path: PathLike,

options:

| (ObjectEncodingOptions & {

withFileTypes?: false | undefined;

})

| BufferEncoding

| undefined

| null,

callback: (err: NodeJS.ErrnoException | null, files: string[] | Buffer[]) => void

): void;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export function readdir(path: PathLike, callback: (err: NodeJS.ErrnoException | null, files: string[]) => void): void;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options If called with `withFileTypes: true` the result data will be an array of Dirent.

\*/

export function readdir(

path: PathLike,

options: ObjectEncodingOptions & {

withFileTypes: true;

},

callback: (err: NodeJS.ErrnoException | null, files: Dirent[]) => void

): void;

export namespace readdir {

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(

path: PathLike,

options?:

| {

encoding: BufferEncoding | null;

withFileTypes?: false | undefined;

}

| BufferEncoding

| null

): Promise<string[]>;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(

path: PathLike,

options:

| 'buffer'

| {

encoding: 'buffer';

withFileTypes?: false | undefined;

}

): Promise<Buffer[]>;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

function \_\_promisify\_\_(

path: PathLike,

options?:

| (ObjectEncodingOptions & {

withFileTypes?: false | undefined;

})

| BufferEncoding

| null

): Promise<string[] | Buffer[]>;

/\*\*

\* Asynchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options If called with `withFileTypes: true` the result data will be an array of Dirent

\*/

function \_\_promisify\_\_(

path: PathLike,

options: ObjectEncodingOptions & {

withFileTypes: true;

}

): Promise<Dirent[]>;

}

/\*\*

\* Reads the contents of the directory.

\*

\* See the POSIX [`readdir(3)`](http://man7.org/linux/man-pages/man3/readdir.3.html) documentation for more details.

\*

\* The optional `options` argument can be a string specifying an encoding, or an

\* object with an `encoding` property specifying the character encoding to use for

\* the filenames returned. If the `encoding` is set to `'buffer'`,

\* the filenames returned will be passed as `Buffer` objects.

\*

\* If `options.withFileTypes` is set to `true`, the result will contain `fs.Dirent` objects.

\* @since v0.1.21

\*/

export function readdirSync(

path: PathLike,

options?:

| {

encoding: BufferEncoding | null;

withFileTypes?: false | undefined;

}

| BufferEncoding

| null

): string[];

/\*\*

\* Synchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readdirSync(

path: PathLike,

options:

| {

encoding: 'buffer';

withFileTypes?: false | undefined;

}

| 'buffer'

): Buffer[];

/\*\*

\* Synchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options The encoding (or an object specifying the encoding), used as the encoding of the result. If not provided, `'utf8'` is used.

\*/

export function readdirSync(

path: PathLike,

options?:

| (ObjectEncodingOptions & {

withFileTypes?: false | undefined;

})

| BufferEncoding

| null

): string[] | Buffer[];

/\*\*

\* Synchronous readdir(3) - read a directory.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param options If called with `withFileTypes: true` the result data will be an array of Dirent.

\*/

export function readdirSync(

path: PathLike,

options: ObjectEncodingOptions & {

withFileTypes: true;

}

): Dirent[];

/\*\*

\* Closes the file descriptor. No arguments other than a possible exception are

\* given to the completion callback.

\*

\* Calling `fs.close()` on any file descriptor (`fd`) that is currently in use

\* through any other `fs` operation may lead to undefined behavior.

\*

\* See the POSIX [`close(2)`](http://man7.org/linux/man-pages/man2/close.2.html) documentation for more detail.

\* @since v0.0.2

\*/

export function close(fd: number, callback?: NoParamCallback): void;

export namespace close {

/\*\*

\* Asynchronous close(2) - close a file descriptor.

\* @param fd A file descriptor.

\*/

function \_\_promisify\_\_(fd: number): Promise<void>;

}

/\*\*

\* Closes the file descriptor. Returns `undefined`.

\*

\* Calling `fs.closeSync()` on any file descriptor (`fd`) that is currently in use

\* through any other `fs` operation may lead to undefined behavior.

\*

\* See the POSIX [`close(2)`](http://man7.org/linux/man-pages/man2/close.2.html) documentation for more detail.

\* @since v0.1.21

\*/

export function closeSync(fd: number): void;

/\*\*

\* Asynchronous file open. See the POSIX [`open(2)`](http://man7.org/linux/man-pages/man2/open.2.html) documentation for more details.

\*

\* `mode` sets the file mode (permission and sticky bits), but only if the file was

\* created. On Windows, only the write permission can be manipulated; see {@link chmod}.

\*

\* The callback gets two arguments `(err, fd)`.

\*

\* Some characters (`< > : " / \ | ? \*`) are reserved under Windows as documented

\* by [Naming Files, Paths, and Namespaces](https://docs.microsoft.com/en-us/windows/desktop/FileIO/naming-a-file). Under NTFS, if the filename contains

\* a colon, Node.js will open a file system stream, as described by [this MSDN page](https://docs.microsoft.com/en-us/windows/desktop/FileIO/using-streams).

\*

\* Functions based on `fs.open()` exhibit this behavior as well:`fs.writeFile()`, `fs.readFile()`, etc.

\* @since v0.0.2

\* @param [flags='r'] See `support of file system `flags``.

\* @param [mode=0o666]

\*/

export function open(path: PathLike, flags: OpenMode, mode: Mode | undefined | null, callback: (err: NodeJS.ErrnoException | null, fd: number) => void): void;

/\*\*

\* Asynchronous open(2) - open and possibly create a file. If the file is created, its mode will be `0o666`.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\*/

export function open(path: PathLike, flags: OpenMode, callback: (err: NodeJS.ErrnoException | null, fd: number) => void): void;

export namespace open {

/\*\*

\* Asynchronous open(2) - open and possibly create a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param mode A file mode. If a string is passed, it is parsed as an octal integer. If not supplied, defaults to `0o666`.

\*/

function \_\_promisify\_\_(path: PathLike, flags: OpenMode, mode?: Mode | null): Promise<number>;

}

/\*\*

\* Returns an integer representing the file descriptor.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link open}.

\* @since v0.1.21

\* @param [flags='r']

\* @param [mode=0o666]

\*/

export function openSync(path: PathLike, flags: OpenMode, mode?: Mode | null): number;

/\*\*

\* Change the file system timestamps of the object referenced by `path`.

\*

\* The `atime` and `mtime` arguments follow these rules:

\*

\* \* Values can be either numbers representing Unix epoch time in seconds,`Date`s, or a numeric string like `'123456789.0'`.

\* \* If the value can not be converted to a number, or is `NaN`, `Infinity` or`-Infinity`, an `Error` will be thrown.

\* @since v0.4.2

\*/

export function utimes(path: PathLike, atime: TimeLike, mtime: TimeLike, callback: NoParamCallback): void;

export namespace utimes {

/\*\*

\* Asynchronously change file timestamps of the file referenced by the supplied path.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* @param atime The last access time. If a string is provided, it will be coerced to number.

\* @param mtime The last modified time. If a string is provided, it will be coerced to number.

\*/

function \_\_promisify\_\_(path: PathLike, atime: TimeLike, mtime: TimeLike): Promise<void>;

}

/\*\*

\* Returns `undefined`.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link utimes}.

\* @since v0.4.2

\*/

export function utimesSync(path: PathLike, atime: TimeLike, mtime: TimeLike): void;

/\*\*

\* Change the file system timestamps of the object referenced by the supplied file

\* descriptor. See {@link utimes}.

\* @since v0.4.2

\*/

export function futimes(fd: number, atime: TimeLike, mtime: TimeLike, callback: NoParamCallback): void;

export namespace futimes {

/\*\*

\* Asynchronously change file timestamps of the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param atime The last access time. If a string is provided, it will be coerced to number.

\* @param mtime The last modified time. If a string is provided, it will be coerced to number.

\*/

function \_\_promisify\_\_(fd: number, atime: TimeLike, mtime: TimeLike): Promise<void>;

}

/\*\*

\* Synchronous version of {@link futimes}. Returns `undefined`.

\* @since v0.4.2

\*/

export function futimesSync(fd: number, atime: TimeLike, mtime: TimeLike): void;

/\*\*

\* Request that all data for the open file descriptor is flushed to the storage

\* device. The specific implementation is operating system and device specific.

\* Refer to the POSIX [`fsync(2)`](http://man7.org/linux/man-pages/man2/fsync.2.html) documentation for more detail. No arguments other

\* than a possible exception are given to the completion callback.

\* @since v0.1.96

\*/

export function fsync(fd: number, callback: NoParamCallback): void;

export namespace fsync {

/\*\*

\* Asynchronous fsync(2) - synchronize a file's in-core state with the underlying storage device.

\* @param fd A file descriptor.

\*/

function \_\_promisify\_\_(fd: number): Promise<void>;

}

/\*\*

\* Request that all data for the open file descriptor is flushed to the storage

\* device. The specific implementation is operating system and device specific.

\* Refer to the POSIX [`fsync(2)`](http://man7.org/linux/man-pages/man2/fsync.2.html) documentation for more detail. Returns `undefined`.

\* @since v0.1.96

\*/

export function fsyncSync(fd: number): void;

/\*\*

\* Write `buffer` to the file specified by `fd`. If `buffer` is a normal object, it

\* must have an own `toString` function property.

\*

\* `offset` determines the part of the buffer to be written, and `length` is

\* an integer specifying the number of bytes to write.

\*

\* `position` refers to the offset from the beginning of the file where this data

\* should be written. If `typeof position !== 'number'`, the data will be written

\* at the current position. See [`pwrite(2)`](http://man7.org/linux/man-pages/man2/pwrite.2.html).

\*

\* The callback will be given three arguments `(err, bytesWritten, buffer)` where`bytesWritten` specifies how many \_bytes\_ were written from `buffer`.

\*

\* If this method is invoked as its `util.promisify()` ed version, it returns

\* a promise for an `Object` with `bytesWritten` and `buffer` properties.

\*

\* It is unsafe to use `fs.write()` multiple times on the same file without waiting

\* for the callback. For this scenario, {@link createWriteStream} is

\* recommended.

\*

\* On Linux, positional writes don't work when the file is opened in append mode.

\* The kernel ignores the position argument and always appends the data to

\* the end of the file.

\* @since v0.0.2

\*/

export function write<TBuffer extends NodeJS.ArrayBufferView>(

fd: number,

buffer: TBuffer,

offset: number | undefined | null,

length: number | undefined | null,

position: number | undefined | null,

callback: (err: NodeJS.ErrnoException | null, written: number, buffer: TBuffer) => void

): void;

/\*\*

\* Asynchronously writes `buffer` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param offset The part of the buffer to be written. If not supplied, defaults to `0`.

\* @param length The number of bytes to write. If not supplied, defaults to `buffer.length - offset`.

\*/

export function write<TBuffer extends NodeJS.ArrayBufferView>(

fd: number,

buffer: TBuffer,

offset: number | undefined | null,

length: number | undefined | null,

callback: (err: NodeJS.ErrnoException | null, written: number, buffer: TBuffer) => void

): void;

/\*\*

\* Asynchronously writes `buffer` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param offset The part of the buffer to be written. If not supplied, defaults to `0`.

\*/

export function write<TBuffer extends NodeJS.ArrayBufferView>(

fd: number,

buffer: TBuffer,

offset: number | undefined | null,

callback: (err: NodeJS.ErrnoException | null, written: number, buffer: TBuffer) => void

): void;

/\*\*

\* Asynchronously writes `buffer` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\*/

export function write<TBuffer extends NodeJS.ArrayBufferView>(fd: number, buffer: TBuffer, callback: (err: NodeJS.ErrnoException | null, written: number, buffer: TBuffer) => void): void;

/\*\*

\* Asynchronously writes `string` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param string A string to write.

\* @param position The offset from the beginning of the file where this data should be written. If not supplied, defaults to the current position.

\* @param encoding The expected string encoding.

\*/

export function write(

fd: number,

string: string,

position: number | undefined | null,

encoding: BufferEncoding | undefined | null,

callback: (err: NodeJS.ErrnoException | null, written: number, str: string) => void

): void;

/\*\*

\* Asynchronously writes `string` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param string A string to write.

\* @param position The offset from the beginning of the file where this data should be written. If not supplied, defaults to the current position.

\*/

export function write(fd: number, string: string, position: number | undefined | null, callback: (err: NodeJS.ErrnoException | null, written: number, str: string) => void): void;

/\*\*

\* Asynchronously writes `string` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param string A string to write.

\*/

export function write(fd: number, string: string, callback: (err: NodeJS.ErrnoException | null, written: number, str: string) => void): void;

export namespace write {

/\*\*

\* Asynchronously writes `buffer` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param offset The part of the buffer to be written. If not supplied, defaults to `0`.

\* @param length The number of bytes to write. If not supplied, defaults to `buffer.length - offset`.

\* @param position The offset from the beginning of the file where this data should be written. If not supplied, defaults to the current position.

\*/

function \_\_promisify\_\_<TBuffer extends NodeJS.ArrayBufferView>(

fd: number,

buffer?: TBuffer,

offset?: number,

length?: number,

position?: number | null

): Promise<{

bytesWritten: number;

buffer: TBuffer;

}>;

/\*\*

\* Asynchronously writes `string` to the file referenced by the supplied file descriptor.

\* @param fd A file descriptor.

\* @param string A string to write.

\* @param position The offset from the beginning of the file where this data should be written. If not supplied, defaults to the current position.

\* @param encoding The expected string encoding.

\*/

function \_\_promisify\_\_(

fd: number,

string: string,

position?: number | null,

encoding?: BufferEncoding | null

): Promise<{

bytesWritten: number;

buffer: string;

}>;

}

/\*\*

\* If `buffer` is a plain object, it must have an own (not inherited) `toString`function property.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link write}.

\* @since v0.1.21

\* @return The number of bytes written.

\*/

export function writeSync(fd: number, buffer: NodeJS.ArrayBufferView, offset?: number | null, length?: number | null, position?: number | null): number;

/\*\*

\* Synchronously writes `string` to the file referenced by the supplied file descriptor, returning the number of bytes written.

\* @param fd A file descriptor.

\* @param string A string to write.

\* @param position The offset from the beginning of the file where this data should be written. If not supplied, defaults to the current position.

\* @param encoding The expected string encoding.

\*/

export function writeSync(fd: number, string: string, position?: number | null, encoding?: BufferEncoding | null): number;

export type ReadPosition = number | bigint;

/\*\*

\* Read data from the file specified by `fd`.

\*

\* The callback is given the three arguments, `(err, bytesRead, buffer)`.

\*

\* If the file is not modified concurrently, the end-of-file is reached when the

\* number of bytes read is zero.

\*

\* If this method is invoked as its `util.promisify()` ed version, it returns

\* a promise for an `Object` with `bytesRead` and `buffer` properties.

\* @since v0.0.2

\* @param buffer The buffer that the data will be written to.

\* @param offset The position in `buffer` to write the data to.

\* @param length The number of bytes to read.

\* @param position Specifies where to begin reading from in the file. If `position` is `null` or `-1 `, data will be read from the current file position, and the file position will be updated. If

\* `position` is an integer, the file position will be unchanged.

\*/

export function read<TBuffer extends NodeJS.ArrayBufferView>(

fd: number,

buffer: TBuffer,

offset: number,

length: number,

position: ReadPosition | null,

callback: (err: NodeJS.ErrnoException | null, bytesRead: number, buffer: TBuffer) => void

): void;

export namespace read {

/\*\*

\* @param fd A file descriptor.

\* @param buffer The buffer that the data will be written to.

\* @param offset The offset in the buffer at which to start writing.

\* @param length The number of bytes to read.

\* @param position The offset from the beginning of the file from which data should be read. If `null`, data will be read from the current position.

\*/

function \_\_promisify\_\_<TBuffer extends NodeJS.ArrayBufferView>(

fd: number,

buffer: TBuffer,

offset: number,

length: number,

position: number | null

): Promise<{

bytesRead: number;

buffer: TBuffer;

}>;

}

export interface ReadSyncOptions {

/\*\*

\* @default 0

\*/

offset?: number | undefined;

/\*\*

\* @default `length of buffer`

\*/

length?: number | undefined;

/\*\*

\* @default null

\*/

position?: ReadPosition | null | undefined;

}

/\*\*

\* Returns the number of `bytesRead`.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link read}.

\* @since v0.1.21

\*/

export function readSync(fd: number, buffer: NodeJS.ArrayBufferView, offset: number, length: number, position: ReadPosition | null): number;

/\*\*

\* Similar to the above `fs.readSync` function, this version takes an optional `options` object.

\* If no `options` object is specified, it will default with the above values.

\*/

export function readSync(fd: number, buffer: NodeJS.ArrayBufferView, opts?: ReadSyncOptions): number;

/\*\*

\* Asynchronously reads the entire contents of a file.

\*

\* ```js

\* import { readFile } from 'fs';

\*

\* readFile('/etc/passwd', (err, data) => {

\* if (err) throw err;

\* console.log(data);

\* });

\* ```

\*

\* The callback is passed two arguments `(err, data)`, where `data` is the

\* contents of the file.

\*

\* If no encoding is specified, then the raw buffer is returned.

\*

\* If `options` is a string, then it specifies the encoding:

\*

\* ```js

\* import { readFile } from 'fs';

\*

\* readFile('/etc/passwd', 'utf8', callback);

\* ```

\*

\* When the path is a directory, the behavior of `fs.readFile()` and {@link readFileSync} is platform-specific. On macOS, Linux, and Windows, an

\* error will be returned. On FreeBSD, a representation of the directory's contents

\* will be returned.

\*

\* ```js

\* import { readFile } from 'fs';

\*

\* // macOS, Linux, and Windows

\* readFile('<directory>', (err, data) => {

\* // => [Error: EISDIR: illegal operation on a directory, read <directory>]

\* });

\*

\* // FreeBSD

\* readFile('<directory>', (err, data) => {

\* // => null, <data>

\* });

\* ```

\*

\* It is possible to abort an ongoing request using an `AbortSignal`. If a

\* request is aborted the callback is called with an `AbortError`:

\*

\* ```js

\* import { readFile } from 'fs';

\*

\* const controller = new AbortController();

\* const signal = controller.signal;

\* readFile(fileInfo[0].name, { signal }, (err, buf) => {

\* // ...

\* });

\* // When you want to abort the request

\* controller.abort();

\* ```

\*

\* The `fs.readFile()` function buffers the entire file. To minimize memory costs,

\* when possible prefer streaming via `fs.createReadStream()`.

\*

\* Aborting an ongoing request does not abort individual operating

\* system requests but rather the internal buffering `fs.readFile` performs.

\* @since v0.1.29

\* @param path filename or file descriptor

\*/

export function readFile(

path: PathOrFileDescriptor,

options:

| ({

encoding?: null | undefined;

flag?: string | undefined;

} & Abortable)

| undefined

| null,

callback: (err: NodeJS.ErrnoException | null, data: Buffer) => void

): void;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options Either the encoding for the result, or an object that contains the encoding and an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

export function readFile(

path: PathOrFileDescriptor,

options:

| ({

encoding: BufferEncoding;

flag?: string | undefined;

} & Abortable)

| BufferEncoding,

callback: (err: NodeJS.ErrnoException | null, data: string) => void

): void;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options Either the encoding for the result, or an object that contains the encoding and an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

export function readFile(

path: PathOrFileDescriptor,

options:

| (ObjectEncodingOptions & {

flag?: string | undefined;

} & Abortable)

| BufferEncoding

| undefined

| null,

callback: (err: NodeJS.ErrnoException | null, data: string | Buffer) => void

): void;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\*/

export function readFile(path: PathOrFileDescriptor, callback: (err: NodeJS.ErrnoException | null, data: Buffer) => void): void;

export namespace readFile {

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options An object that may contain an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

function \_\_promisify\_\_(

path: PathOrFileDescriptor,

options?: {

encoding?: null | undefined;

flag?: string | undefined;

} | null

): Promise<Buffer>;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options Either the encoding for the result, or an object that contains the encoding and an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

function \_\_promisify\_\_(

path: PathOrFileDescriptor,

options:

| {

encoding: BufferEncoding;

flag?: string | undefined;

}

| BufferEncoding

): Promise<string>;

/\*\*

\* Asynchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options Either the encoding for the result, or an object that contains the encoding and an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

function \_\_promisify\_\_(

path: PathOrFileDescriptor,

options?:

| (ObjectEncodingOptions & {

flag?: string | undefined;

})

| BufferEncoding

| null

): Promise<string | Buffer>;

}

/\*\*

\* Returns the contents of the `path`.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link readFile}.

\*

\* If the `encoding` option is specified then this function returns a

\* string. Otherwise it returns a buffer.

\*

\* Similar to {@link readFile}, when the path is a directory, the behavior of`fs.readFileSync()` is platform-specific.

\*

\* ```js

\* import { readFileSync } from 'fs';

\*

\* // macOS, Linux, and Windows

\* readFileSync('<directory>');

\* // => [Error: EISDIR: illegal operation on a directory, read <directory>]

\*

\* // FreeBSD

\* readFileSync('<directory>'); // => <data>

\* ```

\* @since v0.1.8

\* @param path filename or file descriptor

\*/

export function readFileSync(

path: PathOrFileDescriptor,

options?: {

encoding?: null | undefined;

flag?: string | undefined;

} | null

): Buffer;

/\*\*

\* Synchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options Either the encoding for the result, or an object that contains the encoding and an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

export function readFileSync(

path: PathOrFileDescriptor,

options:

| {

encoding: BufferEncoding;

flag?: string | undefined;

}

| BufferEncoding

): string;

/\*\*

\* Synchronously reads the entire contents of a file.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param options Either the encoding for the result, or an object that contains the encoding and an optional flag.

\* If a flag is not provided, it defaults to `'r'`.

\*/

export function readFileSync(

path: PathOrFileDescriptor,

options?:

| (ObjectEncodingOptions & {

flag?: string | undefined;

})

| BufferEncoding

| null

): string | Buffer;

export type WriteFileOptions =

| (ObjectEncodingOptions &

Abortable & {

mode?: Mode | undefined;

flag?: string | undefined;

})

| BufferEncoding

| null;

/\*\*

\* When `file` is a filename, asynchronously writes data to the file, replacing the

\* file if it already exists. `data` can be a string or a buffer.

\*

\* When `file` is a file descriptor, the behavior is similar to calling`fs.write()` directly (which is recommended). See the notes below on using

\* a file descriptor.

\*

\* The `encoding` option is ignored if `data` is a buffer.

\*

\* The `mode` option only affects the newly created file. See {@link open} for more details.

\*

\* If `data` is a plain object, it must have an own (not inherited) `toString`function property.

\*

\* ```js

\* import { writeFile } from 'fs';

\* import { Buffer } from 'buffer';

\*

\* const data = new Uint8Array(Buffer.from('Hello Node.js'));

\* writeFile('message.txt', data, (err) => {

\* if (err) throw err;

\* console.log('The file has been saved!');

\* });

\* ```

\*

\* If `options` is a string, then it specifies the encoding:

\*

\* ```js

\* import { writeFile } from 'fs';

\*

\* writeFile('message.txt', 'Hello Node.js', 'utf8', callback);

\* ```

\*

\* It is unsafe to use `fs.writeFile()` multiple times on the same file without

\* waiting for the callback. For this scenario, {@link createWriteStream} is

\* recommended.

\*

\* Similarly to `fs.readFile` \- `fs.writeFile` is a convenience method that

\* performs multiple `write` calls internally to write the buffer passed to it.

\* For performance sensitive code consider using {@link createWriteStream}.

\*

\* It is possible to use an `AbortSignal` to cancel an `fs.writeFile()`.

\* Cancelation is "best effort", and some amount of data is likely still

\* to be written.

\*

\* ```js

\* import { writeFile } from 'fs';

\* import { Buffer } from 'buffer';

\*

\* const controller = new AbortController();

\* const { signal } = controller;

\* const data = new Uint8Array(Buffer.from('Hello Node.js'));

\* writeFile('message.txt', data, { signal }, (err) => {

\* // When a request is aborted - the callback is called with an AbortError

\* });

\* // When the request should be aborted

\* controller.abort();

\* ```

\*

\* Aborting an ongoing request does not abort individual operating

\* system requests but rather the internal buffering `fs.writeFile` performs.

\* @since v0.1.29

\* @param file filename or file descriptor

\*/

export function writeFile(file: PathOrFileDescriptor, data: string | NodeJS.ArrayBufferView, options: WriteFileOptions, callback: NoParamCallback): void;

/\*\*

\* Asynchronously writes data to a file, replacing the file if it already exists.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param data The data to write. If something other than a Buffer or Uint8Array is provided, the value is coerced to a string.

\*/

export function writeFile(path: PathOrFileDescriptor, data: string | NodeJS.ArrayBufferView, callback: NoParamCallback): void;

export namespace writeFile {

/\*\*

\* Asynchronously writes data to a file, replacing the file if it already exists.

\* @param path A path to a file. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param data The data to write. If something other than a Buffer or Uint8Array is provided, the value is coerced to a string.

\* @param options Either the encoding for the file, or an object optionally specifying the encoding, file mode, and flag.

\* If `encoding` is not supplied, the default of `'utf8'` is used.

\* If `mode` is not supplied, the default of `0o666` is used.

\* If `mode` is a string, it is parsed as an octal integer.

\* If `flag` is not supplied, the default of `'w'` is used.

\*/

function \_\_promisify\_\_(path: PathOrFileDescriptor, data: string | NodeJS.ArrayBufferView, options?: WriteFileOptions): Promise<void>;

}

/\*\*

\* Returns `undefined`.

\*

\* If `data` is a plain object, it must have an own (not inherited) `toString`function property.

\*

\* The `mode` option only affects the newly created file. See {@link open} for more details.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link writeFile}.

\* @since v0.1.29

\* @param file filename or file descriptor

\*/

export function writeFileSync(file: PathOrFileDescriptor, data: string | NodeJS.ArrayBufferView, options?: WriteFileOptions): void;

/\*\*

\* Asynchronously append data to a file, creating the file if it does not yet

\* exist. `data` can be a string or a `Buffer`.

\*

\* The `mode` option only affects the newly created file. See {@link open} for more details.

\*

\* ```js

\* import { appendFile } from 'fs';

\*

\* appendFile('message.txt', 'data to append', (err) => {

\* if (err) throw err;

\* console.log('The "data to append" was appended to file!');

\* });

\* ```

\*

\* If `options` is a string, then it specifies the encoding:

\*

\* ```js

\* import { appendFile } from 'fs';

\*

\* appendFile('message.txt', 'data to append', 'utf8', callback);

\* ```

\*

\* The `path` may be specified as a numeric file descriptor that has been opened

\* for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will

\* not be closed automatically.

\*

\* ```js

\* import { open, close, appendFile } from 'fs';

\*

\* function closeFd(fd) {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\*

\* open('message.txt', 'a', (err, fd) => {

\* if (err) throw err;

\*

\* try {

\* appendFile(fd, 'data to append', 'utf8', (err) => {

\* closeFd(fd);

\* if (err) throw err;

\* });

\* } catch (err) {

\* closeFd(fd);

\* throw err;

\* }

\* });

\* ```

\* @since v0.6.7

\* @param path filename or file descriptor

\*/

export function appendFile(path: PathOrFileDescriptor, data: string | Uint8Array, options: WriteFileOptions, callback: NoParamCallback): void;

/\*\*

\* Asynchronously append data to a file, creating the file if it does not exist.

\* @param file A path to a file. If a URL is provided, it must use the `file:` protocol.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param data The data to write. If something other than a Buffer or Uint8Array is provided, the value is coerced to a string.

\*/

export function appendFile(file: PathOrFileDescriptor, data: string | Uint8Array, callback: NoParamCallback): void;

export namespace appendFile {

/\*\*

\* Asynchronously append data to a file, creating the file if it does not exist.

\* @param file A path to a file. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\* If a file descriptor is provided, the underlying file will \_not\_ be closed automatically.

\* @param data The data to write. If something other than a Buffer or Uint8Array is provided, the value is coerced to a string.

\* @param options Either the encoding for the file, or an object optionally specifying the encoding, file mode, and flag.

\* If `encoding` is not supplied, the default of `'utf8'` is used.

\* If `mode` is not supplied, the default of `0o666` is used.

\* If `mode` is a string, it is parsed as an octal integer.

\* If `flag` is not supplied, the default of `'a'` is used.

\*/

function \_\_promisify\_\_(file: PathOrFileDescriptor, data: string | Uint8Array, options?: WriteFileOptions): Promise<void>;

}

/\*\*

\* Synchronously append data to a file, creating the file if it does not yet

\* exist. `data` can be a string or a `Buffer`.

\*

\* The `mode` option only affects the newly created file. See {@link open} for more details.

\*

\* ```js

\* import { appendFileSync } from 'fs';

\*

\* try {

\* appendFileSync('message.txt', 'data to append');

\* console.log('The "data to append" was appended to file!');

\* } catch (err) {

\* // Handle the error

\* }

\* ```

\*

\* If `options` is a string, then it specifies the encoding:

\*

\* ```js

\* import { appendFileSync } from 'fs';

\*

\* appendFileSync('message.txt', 'data to append', 'utf8');

\* ```

\*

\* The `path` may be specified as a numeric file descriptor that has been opened

\* for appending (using `fs.open()` or `fs.openSync()`). The file descriptor will

\* not be closed automatically.

\*

\* ```js

\* import { openSync, closeSync, appendFileSync } from 'fs';

\*

\* let fd;

\*

\* try {

\* fd = openSync('message.txt', 'a');

\* appendFileSync(fd, 'data to append', 'utf8');

\* } catch (err) {

\* // Handle the error

\* } finally {

\* if (fd !== undefined)

\* closeSync(fd);

\* }

\* ```

\* @since v0.6.7

\* @param path filename or file descriptor

\*/

export function appendFileSync(path: PathOrFileDescriptor, data: string | Uint8Array, options?: WriteFileOptions): void;

/\*\*

\* Watch for changes on `filename`. The callback `listener` will be called each

\* time the file is accessed.

\*

\* The `options` argument may be omitted. If provided, it should be an object. The`options` object may contain a boolean named `persistent` that indicates

\* whether the process should continue to run as long as files are being watched.

\* The `options` object may specify an `interval` property indicating how often the

\* target should be polled in milliseconds.

\*

\* The `listener` gets two arguments the current stat object and the previous

\* stat object:

\*

\* ```js

\* import { watchFile } from 'fs';

\*

\* watchFile('message.text', (curr, prev) => {

\* console.log(`the current mtime is: ${curr.mtime}`);

\* console.log(`the previous mtime was: ${prev.mtime}`);

\* });

\* ```

\*

\* These stat objects are instances of `fs.Stat`. If the `bigint` option is `true`,

\* the numeric values in these objects are specified as `BigInt`s.

\*

\* To be notified when the file was modified, not just accessed, it is necessary

\* to compare `curr.mtimeMs` and `prev.mtimeMs`.

\*

\* When an `fs.watchFile` operation results in an `ENOENT` error, it

\* will invoke the listener once, with all the fields zeroed (or, for dates, the

\* Unix Epoch). If the file is created later on, the listener will be called

\* again, with the latest stat objects. This is a change in functionality since

\* v0.10.

\*

\* Using {@link watch} is more efficient than `fs.watchFile` and`fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and`fs.unwatchFile` when possible.

\*

\* When a file being watched by `fs.watchFile()` disappears and reappears,

\* then the contents of `previous` in the second callback event (the file's

\* reappearance) will be the same as the contents of `previous` in the first

\* callback event (its disappearance).

\*

\* This happens when:

\*

\* \* the file is deleted, followed by a restore

\* \* the file is renamed and then renamed a second time back to its original name

\* @since v0.1.31

\*/

export interface WatchFileOptions {

bigint?: boolean | undefined;

persistent?: boolean | undefined;

interval?: number | undefined;

}

/\*\*

\* Watch for changes on `filename`. The callback `listener` will be called each

\* time the file is accessed.

\*

\* The `options` argument may be omitted. If provided, it should be an object. The`options` object may contain a boolean named `persistent` that indicates

\* whether the process should continue to run as long as files are being watched.

\* The `options` object may specify an `interval` property indicating how often the

\* target should be polled in milliseconds.

\*

\* The `listener` gets two arguments the current stat object and the previous

\* stat object:

\*

\* ```js

\* import { watchFile } from 'fs';

\*

\* watchFile('message.text', (curr, prev) => {

\* console.log(`the current mtime is: ${curr.mtime}`);

\* console.log(`the previous mtime was: ${prev.mtime}`);

\* });

\* ```

\*

\* These stat objects are instances of `fs.Stat`. If the `bigint` option is `true`,

\* the numeric values in these objects are specified as `BigInt`s.

\*

\* To be notified when the file was modified, not just accessed, it is necessary

\* to compare `curr.mtimeMs` and `prev.mtimeMs`.

\*

\* When an `fs.watchFile` operation results in an `ENOENT` error, it

\* will invoke the listener once, with all the fields zeroed (or, for dates, the

\* Unix Epoch). If the file is created later on, the listener will be called

\* again, with the latest stat objects. This is a change in functionality since

\* v0.10.

\*

\* Using {@link watch} is more efficient than `fs.watchFile` and`fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and`fs.unwatchFile` when possible.

\*

\* When a file being watched by `fs.watchFile()` disappears and reappears,

\* then the contents of `previous` in the second callback event (the file's

\* reappearance) will be the same as the contents of `previous` in the first

\* callback event (its disappearance).

\*

\* This happens when:

\*

\* \* the file is deleted, followed by a restore

\* \* the file is renamed and then renamed a second time back to its original name

\* @since v0.1.31

\*/

export function watchFile(

filename: PathLike,

options:

| (WatchFileOptions & {

bigint?: false | undefined;

})

| undefined,

listener: (curr: Stats, prev: Stats) => void

): StatWatcher;

export function watchFile(

filename: PathLike,

options:

| (WatchFileOptions & {

bigint: true;

})

| undefined,

listener: (curr: BigIntStats, prev: BigIntStats) => void

): StatWatcher;

/\*\*

\* Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

\* @param filename A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\*/

export function watchFile(filename: PathLike, listener: (curr: Stats, prev: Stats) => void): StatWatcher;

/\*\*

\* Stop watching for changes on `filename`. If `listener` is specified, only that

\* particular listener is removed. Otherwise, \_all\_ listeners are removed,

\* effectively stopping watching of `filename`.

\*

\* Calling `fs.unwatchFile()` with a filename that is not being watched is a

\* no-op, not an error.

\*

\* Using {@link watch} is more efficient than `fs.watchFile()` and`fs.unwatchFile()`. `fs.watch()` should be used instead of `fs.watchFile()`and `fs.unwatchFile()` when possible.

\* @since v0.1.31

\* @param listener Optional, a listener previously attached using `fs.watchFile()`

\*/

export function unwatchFile(filename: PathLike, listener?: (curr: Stats, prev: Stats) => void): void;

export interface WatchOptions extends Abortable {

encoding?: BufferEncoding | 'buffer' | undefined;

persistent?: boolean | undefined;

recursive?: boolean | undefined;

}

export type WatchEventType = 'rename' | 'change';

export type WatchListener<T> = (event: WatchEventType, filename: T) => void;

/\*\*

\* Watch for changes on `filename`, where `filename` is either a file or a

\* directory.

\*

\* The second argument is optional. If `options` is provided as a string, it

\* specifies the `encoding`. Otherwise `options` should be passed as an object.

\*

\* The listener callback gets two arguments `(eventType, filename)`. `eventType`is either `'rename'` or `'change'`, and `filename` is the name of the file

\* which triggered the event.

\*

\* On most platforms, `'rename'` is emitted whenever a filename appears or

\* disappears in the directory.

\*

\* The listener callback is attached to the `'change'` event fired by `fs.FSWatcher`, but it is not the same thing as the `'change'` value of`eventType`.

\*

\* If a `signal` is passed, aborting the corresponding AbortController will close

\* the returned `fs.FSWatcher`.

\* @since v0.5.10

\* @param listener

\*/

export function watch(

filename: PathLike,

options:

| (WatchOptions & {

encoding: 'buffer';

})

| 'buffer',

listener?: WatchListener<Buffer>

): FSWatcher;

/\*\*

\* Watch for changes on `filename`, where `filename` is either a file or a directory, returning an `FSWatcher`.

\* @param filename A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the encoding for the filename provided to the listener, or an object optionally specifying encoding, persistent, and recursive options.

\* If `encoding` is not supplied, the default of `'utf8'` is used.

\* If `persistent` is not supplied, the default of `true` is used.

\* If `recursive` is not supplied, the default of `false` is used.

\*/

export function watch(filename: PathLike, options?: WatchOptions | BufferEncoding | null, listener?: WatchListener<string>): FSWatcher;

/\*\*

\* Watch for changes on `filename`, where `filename` is either a file or a directory, returning an `FSWatcher`.

\* @param filename A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\* @param options Either the encoding for the filename provided to the listener, or an object optionally specifying encoding, persistent, and recursive options.

\* If `encoding` is not supplied, the default of `'utf8'` is used.

\* If `persistent` is not supplied, the default of `true` is used.

\* If `recursive` is not supplied, the default of `false` is used.

\*/

export function watch(filename: PathLike, options: WatchOptions | string, listener?: WatchListener<string | Buffer>): FSWatcher;

/\*\*

\* Watch for changes on `filename`, where `filename` is either a file or a directory, returning an `FSWatcher`.

\* @param filename A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\*/

export function watch(filename: PathLike, listener?: WatchListener<string>): FSWatcher;

/\*\*

\* Test whether or not the given path exists by checking with the file system.

\* Then call the `callback` argument with either true or false:

\*

\* ```js

\* import { exists } from 'fs';

\*

\* exists('/etc/passwd', (e) => {

\* console.log(e ? 'it exists' : 'no passwd!');

\* });

\* ```

\*

\* \*\*The parameters for this callback are not consistent with other Node.js\*\*

\* \*\*callbacks.\*\* Normally, the first parameter to a Node.js callback is an `err`parameter, optionally followed by other parameters. The `fs.exists()` callback

\* has only one boolean parameter. This is one reason `fs.access()` is recommended

\* instead of `fs.exists()`.

\*

\* Using `fs.exists()` to check for the existence of a file before calling`fs.open()`, `fs.readFile()` or `fs.writeFile()` is not recommended. Doing

\* so introduces a race condition, since other processes may change the file's

\* state between the two calls. Instead, user code should open/read/write the

\* file directly and handle the error raised if the file does not exist.

\*

\* \*\*write (NOT RECOMMENDED)\*\*

\*

\* ```js

\* import { exists, open, close } from 'fs';

\*

\* exists('myfile', (e) => {

\* if (e) {

\* console.error('myfile already exists');

\* } else {

\* open('myfile', 'wx', (err, fd) => {

\* if (err) throw err;

\*

\* try {

\* writeMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* }

\* });

\* ```

\*

\* \*\*write (RECOMMENDED)\*\*

\*

\* ```js

\* import { open, close } from 'fs';

\* open('myfile', 'wx', (err, fd) => {

\* if (err) {

\* if (err.code === 'EEXIST') {

\* console.error('myfile already exists');

\* return;

\* }

\*

\* throw err;

\* }

\*

\* try {

\* writeMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* ```

\*

\* \*\*read (NOT RECOMMENDED)\*\*

\*

\* ```js

\* import { open, close, exists } from 'fs';

\*

\* exists('myfile', (e) => {

\* if (e) {

\* open('myfile', 'r', (err, fd) => {

\* if (err) throw err;

\*

\* try {

\* readMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* } else {

\* console.error('myfile does not exist');

\* }

\* });

\* ```

\*

\* \*\*read (RECOMMENDED)\*\*

\*

\* ```js

\* import { open, close } from 'fs';

\*

\* open('myfile', 'r', (err, fd) => {

\* if (err) {

\* if (err.code === 'ENOENT') {

\* console.error('myfile does not exist');

\* return;

\* }

\*

\* throw err;

\* }

\*

\* try {

\* readMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* ```

\*

\* The "not recommended" examples above check for existence and then use the

\* file; the "recommended" examples are better because they use the file directly

\* and handle the error, if any.

\*

\* In general, check for the existence of a file only if the file won’t be

\* used directly, for example when its existence is a signal from another

\* process.

\* @since v0.0.2

\* @deprecated Since v1.0.0 - Use {@link stat} or {@link access} instead.

\*/

export function exists(path: PathLike, callback: (exists: boolean) => void): void;

/\*\* @deprecated \*/

export namespace exists {

/\*\*

\* @param path A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\*/

function \_\_promisify\_\_(path: PathLike): Promise<boolean>;

}

/\*\*

\* Returns `true` if the path exists, `false` otherwise.

\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link exists}.

\*

\* `fs.exists()` is deprecated, but `fs.existsSync()` is not. The `callback`parameter to `fs.exists()` accepts parameters that are inconsistent with other

\* Node.js callbacks. `fs.existsSync()` does not use a callback.

\*

\* ```js

\* import { existsSync } from 'fs';

\*

\* if (existsSync('/etc/passwd'))

\* console.log('The path exists.');

\* ```

\* @since v0.1.21

\*/

export function existsSync(path: PathLike): boolean;

export namespace constants {

// File Access Constants

/\*\* Constant for fs.access(). File is visible to the calling process. \*/

const F\_OK: number;

/\*\* Constant for fs.access(). File can be read by the calling process. \*/

const R\_OK: number;

/\*\* Constant for fs.access(). File can be written by the calling process. \*/

const W\_OK: number;

/\*\* Constant for fs.access(). File can be executed by the calling process. \*/

const X\_OK: number;

// File Copy Constants

/\*\* Constant for fs.copyFile. Flag indicating the destination file should not be overwritten if it already exists. \*/

const COPYFILE\_EXCL: number;

/\*\*

\* Constant for fs.copyFile. copy operation will attempt to create a copy-on-write reflink.

\* If the underlying platform does not support copy-on-write, then a fallback copy mechanism is used.

\*/

const COPYFILE\_FICLONE: number;

/\*\*

\* Constant for fs.copyFile. Copy operation will attempt to create a copy-on-write reflink.

\* If the underlying platform does not support copy-on-write, then the operation will fail with an error.

\*/

const COPYFILE\_FICLONE\_FORCE: number;

// File Open Constants

/\*\* Constant for fs.open(). Flag indicating to open a file for read-only access. \*/

const O\_RDONLY: number;

/\*\* Constant for fs.open(). Flag indicating to open a file for write-only access. \*/

const O\_WRONLY: number;

/\*\* Constant for fs.open(). Flag indicating to open a file for read-write access. \*/

const O\_RDWR: number;

/\*\* Constant for fs.open(). Flag indicating to create the file if it does not already exist. \*/

const O\_CREAT: number;

/\*\* Constant for fs.open(). Flag indicating that opening a file should fail if the O\_CREAT flag is set and the file already exists. \*/

const O\_EXCL: number;

/\*\*

\* Constant for fs.open(). Flag indicating that if path identifies a terminal device,

\* opening the path shall not cause that terminal to become the controlling terminal for the process

\* (if the process does not already have one).

\*/

const O\_NOCTTY: number;

/\*\* Constant for fs.open(). Flag indicating that if the file exists and is a regular file, and the file is opened successfully for write access, its length shall be truncated to zero. \*/

const O\_TRUNC: number;

/\*\* Constant for fs.open(). Flag indicating that data will be appended to the end of the file. \*/

const O\_APPEND: number;

/\*\* Constant for fs.open(). Flag indicating that the open should fail if the path is not a directory. \*/

const O\_DIRECTORY: number;

/\*\*

\* constant for fs.open().

\* Flag indicating reading accesses to the file system will no longer result in

\* an update to the atime information associated with the file.

\* This flag is available on Linux operating systems only.

\*/

const O\_NOATIME: number;

/\*\* Constant for fs.open(). Flag indicating that the open should fail if the path is a symbolic link. \*/

const O\_NOFOLLOW: number;

/\*\* Constant for fs.open(). Flag indicating that the file is opened for synchronous I/O. \*/

const O\_SYNC: number;

/\*\* Constant for fs.open(). Flag indicating that the file is opened for synchronous I/O with write operations waiting for data integrity. \*/

const O\_DSYNC: number;

/\*\* Constant for fs.open(). Flag indicating to open the symbolic link itself rather than the resource it is pointing to. \*/

const O\_SYMLINK: number;

/\*\* Constant for fs.open(). When set, an attempt will be made to minimize caching effects of file I/O. \*/

const O\_DIRECT: number;

/\*\* Constant for fs.open(). Flag indicating to open the file in nonblocking mode when possible. \*/

const O\_NONBLOCK: number;

// File Type Constants

/\*\* Constant for fs.Stats mode property for determining a file's type. Bit mask used to extract the file type code. \*/

const S\_IFMT: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a regular file. \*/

const S\_IFREG: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a directory. \*/

const S\_IFDIR: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a character-oriented device file. \*/

const S\_IFCHR: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a block-oriented device file. \*/

const S\_IFBLK: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a FIFO/pipe. \*/

const S\_IFIFO: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a symbolic link. \*/

const S\_IFLNK: number;

/\*\* Constant for fs.Stats mode property for determining a file's type. File type constant for a socket. \*/

const S\_IFSOCK: number;

// File Mode Constants

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating readable, writable and executable by owner. \*/

const S\_IRWXU: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating readable by owner. \*/

const S\_IRUSR: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating writable by owner. \*/

const S\_IWUSR: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating executable by owner. \*/

const S\_IXUSR: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating readable, writable and executable by group. \*/

const S\_IRWXG: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating readable by group. \*/

const S\_IRGRP: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating writable by group. \*/

const S\_IWGRP: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating executable by group. \*/

const S\_IXGRP: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating readable, writable and executable by others. \*/

const S\_IRWXO: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating readable by others. \*/

const S\_IROTH: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating writable by others. \*/

const S\_IWOTH: number;

/\*\* Constant for fs.Stats mode property for determining access permissions for a file. File mode indicating executable by others. \*/

const S\_IXOTH: number;

/\*\*

\* When set, a memory file mapping is used to access the file. This flag

\* is available on Windows operating systems only. On other operating systems,

\* this flag is ignored.

\*/

const UV\_FS\_O\_FILEMAP: number;

}

/\*\*

\* Tests a user's permissions for the file or directory specified by `path`.

\* The `mode` argument is an optional integer that specifies the accessibility

\* checks to be performed. Check `File access constants` for possible values

\* of `mode`. It is possible to create a mask consisting of the bitwise OR of

\* two or more values (e.g. `fs.constants.W\_OK | fs.constants.R\_OK`).

\*

\* The final argument, `callback`, is a callback function that is invoked with

\* a possible error argument. If any of the accessibility checks fail, the error

\* argument will be an `Error` object. The following examples check if`package.json` exists, and if it is readable or writable.

\*

\* ```js

\* import { access, constants } from 'fs';

\*

\* const file = 'package.json';

\*

\* // Check if the file exists in the current directory.

\* access(file, constants.F\_OK, (err) => {

\* console.log(`${file} ${err ? 'does not exist' : 'exists'}`);

\* });

\*

\* // Check if the file is readable.

\* access(file, constants.R\_OK, (err) => {

\* console.log(`${file} ${err ? 'is not readable' : 'is readable'}`);

\* });

\*

\* // Check if the file is writable.

\* access(file, constants.W\_OK, (err) => {

\* console.log(`${file} ${err ? 'is not writable' : 'is writable'}`);

\* });

\*

\* // Check if the file exists in the current directory, and if it is writable.

\* access(file, constants.F\_OK | constants.W\_OK, (err) => {

\* if (err) {

\* console.error(

\* `${file} ${err.code === 'ENOENT' ? 'does not exist' : 'is read-only'}`);

\* } else {

\* console.log(`${file} exists, and it is writable`);

\* }

\* });

\* ```

\*

\* Do not use `fs.access()` to check for the accessibility of a file before calling`fs.open()`, `fs.readFile()` or `fs.writeFile()`. Doing

\* so introduces a race condition, since other processes may change the file's

\* state between the two calls. Instead, user code should open/read/write the

\* file directly and handle the error raised if the file is not accessible.

\*

\* \*\*write (NOT RECOMMENDED)\*\*

\*

\* ```js

\* import { access, open, close } from 'fs';

\*

\* access('myfile', (err) => {

\* if (!err) {

\* console.error('myfile already exists');

\* return;

\* }

\*

\* open('myfile', 'wx', (err, fd) => {

\* if (err) throw err;

\*

\* try {

\* writeMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* });

\* ```

\*

\* \*\*write (RECOMMENDED)\*\*

\*

\* ```js

\* import { open, close } from 'fs';

\*

\* open('myfile', 'wx', (err, fd) => {

\* if (err) {

\* if (err.code === 'EEXIST') {

\* console.error('myfile already exists');

\* return;

\* }

\*

\* throw err;

\* }

\*

\* try {

\* writeMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* ```

\*

\* \*\*read (NOT RECOMMENDED)\*\*

\*

\* ```js

\* import { access, open, close } from 'fs';

\* access('myfile', (err) => {

\* if (err) {

\* if (err.code === 'ENOENT') {

\* console.error('myfile does not exist');

\* return;

\* }

\*

\* throw err;

\* }

\*

\* open('myfile', 'r', (err, fd) => {

\* if (err) throw err;

\*

\* try {

\* readMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* });

\* ```

\*

\* \*\*read (RECOMMENDED)\*\*

\*

\* ```js

\* import { open, close } from 'fs';

\*

\* open('myfile', 'r', (err, fd) => {

\* if (err) {

\* if (err.code === 'ENOENT') {

\* console.error('myfile does not exist');

\* return;

\* }

\*

\* throw err;

\* }

\*

\* try {

\* readMyData(fd);

\* } finally {

\* close(fd, (err) => {

\* if (err) throw err;

\* });

\* }

\* });

\* ```

\*

\* The "not recommended" examples above check for accessibility and then use the

\* file; the "recommended" examples are better because they use the file directly

\* and handle the error, if any.

\*

\* In general, check for the accessibility of a file only if the file will not be

\* used directly, for example when its accessibility is a signal from another

\* process.

\*

\* On Windows, access-control policies (ACLs) on a directory may limit access to

\* a file or directory. The `fs.access()` function, however, does not check the

\* ACL and therefore may report that a path is accessible even if the ACL restricts

\* the user from reading or writing to it.

\* @since v0.11.15

\* @param [mode=fs.constants.F\_OK]

\*/

export function access(path: PathLike, mode: number | undefined, callback: NoParamCallback): void;

/\*\*

\* Asynchronously tests a user's permissions for the file specified by path.

\* @param path A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\*/

export function access(path: PathLike, callback: NoParamCallback): void;

export namespace access {

/\*\*

\* Asynchronously tests a user's permissions for the file specified by path.

\* @param path A path to a file or directory. If a URL is provided, it must use the `file:` protocol.

\* URL support is \_experimental\_.

\*/

function \_\_promisify\_\_(path: PathLike, mode?: number): Promise<void>;

}

/\*\*

\* Synchronously tests a user's permissions for the file or directory specified

\* by `path`. The `mode` argument is an optional integer that specifies the

\* accessibility checks to be performed. Check `File access constants` for

\* possible values of `mode`. It is possible to create a mask consisting of

\* the bitwise OR of two or more values

\* (e.g. `fs.constants.W\_OK | fs.constants.R\_OK`).

\*

\* If any of the accessibility checks fail, an `Error` will be thrown. Otherwise,

\* the method will return `undefined`.

\*

\* ```js

\* import { accessSync, constants } from 'fs';

\*

\* try {

\* accessSync('etc/passwd', constants.R\_OK | constants.W\_OK);

\* console.log('can read/write');

\* } catch (err) {

\* console.error('no access!');

\* }

\* ```

\* @since v0.11.15

\* @param [mode=fs.constants.F\_OK]

\*/

export function accessSync(path: PathLike, mode?: number): void;

interface StreamOptions {

flags?: string | undefined;

encoding?: BufferEncoding | undefined;

fd?: number | promises.FileHandle | undefined;

mode?: number | undefined;

autoClose?: boolean | undefined;

/\*\*

\* @default false

\*/

emitClose?: boolean | undefined;

start?: number | undefined;

highWaterMark?: number | undefined;

}

interface ReadStreamOptions extends StreamOptions {

end?: number | undefined;

}

/\*\*

\* Unlike the 16 kb default `highWaterMark` for a `stream.Readable`, the stream

\* returned by this method has a default `highWaterMark` of 64 kb.

\*

\* `options` can include `start` and `end` values to read a range of bytes from

\* the file instead of the entire file. Both `start` and `end` are inclusive and

\* start counting at 0, allowed values are in the

\* \[0, [`Number.MAX\_SAFE\_INTEGER`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Number/MAX\_SAFE\_INTEGER)\] range. If `fd` is specified and `start` is

\* omitted or `undefined`, `fs.createReadStream()` reads sequentially from the

\* current file position. The `encoding` can be any one of those accepted by `Buffer`.

\*

\* If `fd` is specified, `ReadStream` will ignore the `path` argument and will use

\* the specified file descriptor. This means that no `'open'` event will be

\* emitted. `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

\*

\* If `fd` points to a character device that only supports blocking reads

\* (such as keyboard or sound card), read operations do not finish until data is

\* available. This can prevent the process from exiting and the stream from

\* closing naturally.

\*

\* By default, the stream will emit a `'close'` event after it has been

\* destroyed. Set the `emitClose` option to `false` to change this behavior.

\*

\* By providing the `fs` option, it is possible to override the corresponding `fs`implementations for `open`, `read`, and `close`. When providing the `fs` option,

\* an override for `read` is required. If no `fd` is provided, an override for`open` is also required. If `autoClose` is `true`, an override for `close` is

\* also required.

\*

\* ```js

\* import { createReadStream } from 'fs';

\*

\* // Create a stream from some character device.

\* const stream = createReadStream('/dev/input/event0');

\* setTimeout(() => {

\* stream.close(); // This may not close the stream.

\* // Artificially marking end-of-stream, as if the underlying resource had

\* // indicated end-of-file by itself, allows the stream to close.

\* // This does not cancel pending read operations, and if there is such an

\* // operation, the process may still not be able to exit successfully

\* // until it finishes.

\* stream.push(null);

\* stream.read(0);

\* }, 100);

\* ```

\*

\* If `autoClose` is false, then the file descriptor won't be closed, even if

\* there's an error. It is the application's responsibility to close it and make

\* sure there's no file descriptor leak. If `autoClose` is set to true (default

\* behavior), on `'error'` or `'end'` the file descriptor will be closed

\* automatically.

\*

\* `mode` sets the file mode (permission and sticky bits), but only if the

\* file was created.

\*

\* An example to read the last 10 bytes of a file which is 100 bytes long:

\*

\* ```js

\* import { createReadStream } from 'fs';

\*

\* createReadStream('sample.txt', { start: 90, end: 99 });

\* ```

\*

\* If `options` is a string, then it specifies the encoding.

\* @since v0.1.31

\*/

export function createReadStream(path: PathLike, options?: BufferEncoding | ReadStreamOptions): ReadStream;

/\*\*

\* `options` may also include a `start` option to allow writing data at some

\* position past the beginning of the file, allowed values are in the

\* \[0, [`Number.MAX\_SAFE\_INTEGER`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Number/MAX\_SAFE\_INTEGER)\] range. Modifying a file rather than replacing

\* it may require the `flags` option to be set to `r+` rather than the default `w`.

\* The `encoding` can be any one of those accepted by `Buffer`.

\*

\* If `autoClose` is set to true (default behavior) on `'error'` or `'finish'`the file descriptor will be closed automatically. If `autoClose` is false,

\* then the file descriptor won't be closed, even if there's an error.

\* It is the application's responsibility to close it and make sure there's no

\* file descriptor leak.

\*

\* By default, the stream will emit a `'close'` event after it has been

\* destroyed. Set the `emitClose` option to `false` to change this behavior.

\*

\* By providing the `fs` option it is possible to override the corresponding `fs`implementations for `open`, `write`, `writev` and `close`. Overriding `write()`without `writev()` can reduce

\* performance as some optimizations (`\_writev()`)

\* will be disabled. When providing the `fs` option, overrides for at least one of`write` and `writev` are required. If no `fd` option is supplied, an override

\* for `open` is also required. If `autoClose` is `true`, an override for `close`is also required.

\*

\* Like `fs.ReadStream`, if `fd` is specified, `fs.WriteStream` will ignore the`path` argument and will use the specified file descriptor. This means that no`'open'` event will be

\* emitted. `fd` should be blocking; non-blocking `fd`s

\* should be passed to `net.Socket`.

\*

\* If `options` is a string, then it specifies the encoding.

\* @since v0.1.31

\*/

export function createWriteStream(path: PathLike, options?: BufferEncoding | StreamOptions): WriteStream;

/\*\*

\* Forces all currently queued I/O operations associated with the file to the

\* operating system's synchronized I/O completion state. Refer to the POSIX [`fdatasync(2)`](http://man7.org/linux/man-pages/man2/fdatasync.2.html) documentation for details. No arguments other

\* than a possible

\* exception are given to the completion callback.

\* @since v0.1.96

\*/

export function fdatasync(fd: number, callback: NoParamCallback): void;

export namespace fdatasync {

/\*\*

\* Asynchronous fdatasync(2) - synchronize a file's in-core state with storage device.

\* @param fd A file descriptor.

\*/

function \_\_promisify\_\_(fd: number): Promise<void>;

}

/\*\*

\* Forces all currently queued I/O operations associated with the file to the

\* operating system's synchronized I/O completion state. Refer to the POSIX [`fdatasync(2)`](http://man7.org/linux/man-pages/man2/fdatasync.2.html) documentation for details. Returns `undefined`.

\* @since v0.1.96

\*/

export function fdatasyncSync(fd: number): void;

/\*\*

\* Asynchronously copies `src` to `dest`. By default, `dest` is overwritten if it

\* already exists. No arguments other than a possible exception are given to the

\* callback function. Node.js makes no guarantees about the atomicity of the copy

\* operation. If an error occurs after the destination file has been opened for

\* writing, Node.js will attempt to remove the destination.

\*

\* `mode` is an optional integer that specifies the behavior

\* of the copy operation. It is possible to create a mask consisting of the bitwise

\* OR of two or more values (e.g.`fs.constants.COPYFILE\_EXCL | fs.constants.COPYFILE\_FICLONE`).

\*

\* \* `fs.constants.COPYFILE\_EXCL`: The copy operation will fail if `dest` already

\* exists.

\* \* `fs.constants.COPYFILE\_FICLONE`: The copy operation will attempt to create a

\* copy-on-write reflink. If the platform does not support copy-on-write, then a

\* fallback copy mechanism is used.

\* \* `fs.constants.COPYFILE\_FICLONE\_FORCE`: The copy operation will attempt to

\* create a copy-on-write reflink. If the platform does not support

\* copy-on-write, then the operation will fail.

\*

\* ```js

\* import { copyFile, constants } from 'fs';

\*

\* function callback(err) {

\* if (err) throw err;

\* console.log('source.txt was copied to destination.txt');

\* }

\*

\* // destination.txt will be created or overwritten by default.

\* copyFile('source.txt', 'destination.txt', callback);

\*

\* // By using COPYFILE\_EXCL, the operation will fail if destination.txt exists.

\* copyFile('source.txt', 'destination.txt', constants.COPYFILE\_EXCL, callback);

\* ```

\* @since v8.5.0

\* @param src source filename to copy

\* @param dest destination filename of the copy operation

\* @param [mode=0] modifiers for copy operation.

\*/

export function copyFile(src: PathLike, dest: PathLike, callback: NoParamCallback): void;

export function copyFile(src: PathLike, dest: PathLike, mode: number, callback: NoParamCallback): void;

export namespace copyFile {

function \_\_promisify\_\_(src: PathLike, dst: PathLike, mode?: number): Promise<void>;

}

/\*\*

\* Synchronously copies `src` to `dest`. By default, `dest` is overwritten if it

\* already exists. Returns `undefined`. Node.js makes no guarantees about the

\* atomicity of the copy operation. If an error occurs after the destination file

\* has been opened for writing, Node.js will attempt to remove the destination.

\*

\* `mode` is an optional integer that specifies the behavior

\* of the copy operation. It is possible to create a mask consisting of the bitwise

\* OR of two or more values (e.g.`fs.constants.COPYFILE\_EXCL | fs.constants.COPYFILE\_FICLONE`).

\*

\* \* `fs.constants.COPYFILE\_EXCL`: The copy operation will fail if `dest` already

\* exists.

\* \* `fs.constants.COPYFILE\_FICLONE`: The copy operation will attempt to create a

\* copy-on-write reflink. If the platform does not support copy-on-write, then a

\* fallback copy mechanism is used.

\* \* `fs.constants.COPYFILE\_FICLONE\_FORCE`: The copy operation will attempt to

\* create a copy-on-write reflink. If the platform does not support

\* copy-on-write, then the operation will fail.

\*

\* ```js

\* import { copyFileSync, constants } from 'fs';

\*

\* // destination.txt will be created or overwritten by default.

\* copyFileSync('source.txt', 'destination.txt');

\* console.log('source.txt was copied to destination.txt');

\*

\* // By using COPYFILE\_EXCL, the operation will fail if destination.txt exists.

\* copyFileSync('source.txt', 'destination.txt', constants.COPYFILE\_EXCL);

\* ```

\* @since v8.5.0

\* @param src source filename to copy

\* @param dest destination filename of the copy operation

\* @param [mode=0] modifiers for copy operation.

\*/

export function copyFileSync(src: PathLike, dest: PathLike, mode?: number): void;

/\*\*

\* Write an array of `ArrayBufferView`s to the file specified by `fd` using`writev()`.

\*

\* `position` is the offset from the beginning of the file where this data

\* should be written. If `typeof position !== 'number'`, the data will be written

\* at the current position.

\*

\* The callback will be given three arguments: `err`, `bytesWritten`, and`buffers`. `bytesWritten` is how many bytes were written from `buffers`.

\*

\* If this method is `util.promisify()` ed, it returns a promise for an`Object` with `bytesWritten` and `buffers` properties.

\*

\* It is unsafe to use `fs.writev()` multiple times on the same file without

\* waiting for the callback. For this scenario, use {@link createWriteStream}.

\*

\* On Linux, positional writes don't work when the file is opened in append mode.

\* The kernel ignores the position argument and always appends the data to

\* the end of the file.

\* @since v12.9.0

\*/

export function writev(fd: number, buffers: ReadonlyArray<NodeJS.ArrayBufferView>, cb: (err: NodeJS.ErrnoException | null, bytesWritten: number, buffers: NodeJS.ArrayBufferView[]) => void): void;

export function writev(

fd: number,

buffers: ReadonlyArray<NodeJS.ArrayBufferView>,

position: number,

cb: (err: NodeJS.ErrnoException | null, bytesWritten: number, buffers: NodeJS.ArrayBufferView[]) => void

): void;

export interface WriteVResult {

bytesWritten: number;

buffers: NodeJS.ArrayBufferView[];

}

export namespace writev {

function \_\_promisify\_\_(fd: number, buffers: ReadonlyArray<NodeJS.ArrayBufferView>, position?: number): Promise<WriteVResult>;

}

/\*\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link writev}.

\* @since v12.9.0

\* @return The number of bytes written.

\*/

export function writevSync(fd: number, buffers: ReadonlyArray<NodeJS.ArrayBufferView>, position?: number): number;

/\*\*

\* Read from a file specified by `fd` and write to an array of `ArrayBufferView`s

\* using `readv()`.

\*

\* `position` is the offset from the beginning of the file from where data

\* should be read. If `typeof position !== 'number'`, the data will be read

\* from the current position.

\*

\* The callback will be given three arguments: `err`, `bytesRead`, and`buffers`. `bytesRead` is how many bytes were read from the file.

\*

\* If this method is invoked as its `util.promisify()` ed version, it returns

\* a promise for an `Object` with `bytesRead` and `buffers` properties.

\* @since v13.13.0, v12.17.0

\*/

export function readv(fd: number, buffers: ReadonlyArray<NodeJS.ArrayBufferView>, cb: (err: NodeJS.ErrnoException | null, bytesRead: number, buffers: NodeJS.ArrayBufferView[]) => void): void;

export function readv(

fd: number,

buffers: ReadonlyArray<NodeJS.ArrayBufferView>,

position: number,

cb: (err: NodeJS.ErrnoException | null, bytesRead: number, buffers: NodeJS.ArrayBufferView[]) => void

): void;

export interface ReadVResult {

bytesRead: number;

buffers: NodeJS.ArrayBufferView[];

}

export namespace readv {

function \_\_promisify\_\_(fd: number, buffers: ReadonlyArray<NodeJS.ArrayBufferView>, position?: number): Promise<ReadVResult>;

}

/\*\*

\* For detailed information, see the documentation of the asynchronous version of

\* this API: {@link readv}.

\* @since v13.13.0, v12.17.0

\* @return The number of bytes read.

\*/

export function readvSync(fd: number, buffers: ReadonlyArray<NodeJS.ArrayBufferView>, position?: number): number;

export interface OpenDirOptions {

encoding?: BufferEncoding | undefined;

/\*\*

\* Number of directory entries that are buffered

\* internally when reading from the directory. Higher values lead to better

\* performance but higher memory usage.

\* @default 32

\*/

bufferSize?: number | undefined;

}

/\*\*

\* Synchronously open a directory. See [`opendir(3)`](http://man7.org/linux/man-pages/man3/opendir.3.html).

\*

\* Creates an `fs.Dir`, which contains all further functions for reading from

\* and cleaning up the directory.

\*

\* The `encoding` option sets the encoding for the `path` while opening the

\* directory and subsequent read operations.

\* @since v12.12.0

\*/

export function opendirSync(path: PathLike, options?: OpenDirOptions): Dir;

/\*\*

\* Asynchronously open a directory. See the POSIX [`opendir(3)`](http://man7.org/linux/man-pages/man3/opendir.3.html) documentation for

\* more details.

\*

\* Creates an `fs.Dir`, which contains all further functions for reading from

\* and cleaning up the directory.

\*

\* The `encoding` option sets the encoding for the `path` while opening the

\* directory and subsequent read operations.

\* @since v12.12.0

\*/

export function opendir(path: PathLike, cb: (err: NodeJS.ErrnoException | null, dir: Dir) => void): void;

export function opendir(path: PathLike, options: OpenDirOptions, cb: (err: NodeJS.ErrnoException | null, dir: Dir) => void): void;

export namespace opendir {

function \_\_promisify\_\_(path: PathLike, options?: OpenDirOptions): Promise<Dir>;

}

export interface BigIntStats extends StatsBase<bigint> {

atimeNs: bigint;

mtimeNs: bigint;

ctimeNs: bigint;

birthtimeNs: bigint;

}

export interface BigIntOptions {

bigint: true;

}

export interface StatOptions {

bigint?: boolean | undefined;

}

export interface StatSyncOptions extends StatOptions {

throwIfNoEntry?: boolean | undefined;

}

export interface CopyOptions {

/\*\*

\* Dereference symlinks

\* @default false

\*/

dereference?: boolean;

/\*\*

\* When `force` is `false`, and the destination

\* exists, throw an error.

\* @default false

\*/

errorOnExist?: boolean;

/\*\*

\* Function to filter copied files/directories. Return

\* `true` to copy the item, `false` to ignore it.

\*/

filter?(source: string, destination: string): boolean;

/\*\*

\* Overwrite existing file or directory. \_The copy

\* operation will ignore errors if you set this to false and the destination

\* exists. Use the `errorOnExist` option to change this behavior.

\* @default true

\*/

force?: boolean;

/\*\*

\* When `true` timestamps from `src` will

\* be preserved.

\* @default false

\*/

preserveTimestamps?: boolean;

/\*\*

\* Copy directories recursively.

\* @default false

\*/

recursive?: boolean;

}

/\*\*

\* Asynchronously copies the entire directory structure from `src` to `dest`,

\* including subdirectories and files.

\*

\* When copying a directory to another directory, globs are not supported and

\* behavior is similar to `cp dir1/ dir2/`.

\* @since v16.7.0

\* @experimental

\* @param src source path to copy.

\* @param dest destination path to copy to.

\*/

export function cp(source: string, destination: string, callback: (err: NodeJS.ErrnoException | null) => void): void;

export function cp(source: string, destination: string, opts: CopyOptions, callback: (err: NodeJS.ErrnoException | null) => void): void;

/\*\*

\* Synchronously copies the entire directory structure from `src` to `dest`,

\* including subdirectories and files.

\*

\* When copying a directory to another directory, globs are not supported and

\* behavior is similar to `cp dir1/ dir2/`.

\* @since v16.7.0

\* @experimental

\* @param src source path to copy.

\* @param dest destination path to copy to.

\*/

export function cpSync(source: string, destination: string, opts?: CopyOptions): void;

}

declare module 'node:fs' {

export \* from 'fs';

}