



# μCSS

A Node-based CSS and web-graphics processor.

npm package: `gulp-mu-css`

Manual

Version 2.2.2

2026-06-14

## Contents

# 1 Introduction

μCSS is a Node module for processing CSS files and generating web graphics. This manual describes version 2 — the Node-based successor of μCSS 1, introduced in 2013 as an Adobe Photoshop script: from enhanced source stylesheets (`.μ.css`) and the media sources (e.g. PSD drafts), Gulp produces the finished skin files — standard CSS plus all required images, sprites, cursors, fonts and sounds. Image generation is handled by the sibling module μPS, which reproduces the old PS plugins without an Adobe dependency.

Because the μ character keeps causing problems in package and repository names (npm, git), the technical names are `gulp-mu-css` and `gulp-mu-ps` — μCSS and μPS are the display names.

The key features of μCSS 2:

- Source stylesheets stay **syntactically valid CSS** — editors, linters and diff tools work unchanged.
- **Named CSS properties** via skin variables (`$.name`).
- **Computing CSS values with JavaScript expressions** directly in the stylesheet — without substitute characters like `«»j`.
- **Multiple layouts (skins)** from the same sources via manifest files.
- **Automatic sprite atlas generation** including retina variants (`@2x`) and `image-set()`.
- **Cursor management** with hotspot, fallback and retina support.
- **Preloading of CSS images** via a generated preload rule.
- **Automatic image generation** from PSD drafts (buttons, icons, app icons, animation strips) via μPS.
- **Incremental build with cache** — only what changed is regenerated.
- **Meaningful error messages** with file, line and source excerpt.

Unlike μCSS 1, version 2 runs entirely in Node.js (version 18 and up) and needs neither Photoshop nor any other Adobe product. The build is driven via Gulp or directly through the Node API.

## 1.1 Context: Why μCSS?

For almost every individual aspect of μCSS an established single-purpose tool exists — but no system that bundles everything:

Feature	Closest existing equivalent	What is missing there
Variables & color functions	Sass/LESS ( <code>lighten()</code> , <code>mix()</code> ), native CSS ( <code>color-mix()</code> , custom properties)	no embedded JavaScript
JavaScript in CSS values	postcss-functions (registered JS functions as CSS functions)	only named functions — no free expressions, no directives with rule/document access
Stylesheets with full JS power	vanilla-extract (stylesheets-in-TypeScript)	the inverse approach — the regular CSS format is lost
Sprite atlas from CSS references	spritesmith family, postcss-sprites	largely frozen, no retina <code>image-set</code> workflow, no shared cache
Image generation from PSD drafts	only building blocks ( <code>ag-psd</code> , asset export from Figma/Sketch)	no series rendering with layer-style transfer, not coupled to the CSS build
Cursors, preload, skin manifest,	hand-written build scripts	bundled nowhere

Feature	Closest existing equivalent	What is missing there
media cache		

Part of the original 2013  $\mu$ CSS motivation is solved today by native CSS (custom properties, `color-mix()`, nesting) — which is why  $\mu$ CSS 2 deliberately drops LESS support and vendor prefixes. The remaining core is without competition: arbitrary JavaScript expressions in CSS plus directives with AST access, combined with a sprite atlas including retina, a PSD render pipeline and an incremental cache, driven by one manifest per skin. And since  $\mu$ CSS is internally a PostCSS pipeline, the entire PostCSS ecosystem (cssnano, Stylelint, ...) stays attachable instead of competing with it.

## 2 Installation and getting started

μCSS and μPS are available as the npm modules `gulp-mu-css` and `gulp-mu-ps`. Install them in your own project:

```
npm install gulp-mu-css gulp-mu-ps
```

μCSS depends on μPS (atlas and image generation) — not the other way around. Both modules can be used separately.

The typical entry point is a Gulp task that builds a skin manifest:

```
// gulpfile.mjs
import gulp from "gulp";
import { BuildSkin } from "gulp-mu-css";

export async function SkinStd() {
  await BuildSkin("skins/src/std.μcss.mjs");
}
export function SkinWatch() {
  gulp.watch(["skins/src/**", "dev/media/final/**"], SkinStd);
}
```

Running `npx gulp SkinStd` compiles the skin "std" into the directory `skins/std/`. `BuildSkin` is idempotent and cache-backed: a second run without changes finishes in a few milliseconds (see the chapter "Build process").

## 3 Use cases

The following sections show the typical usage scenarios — analogous to the use cases of the old  $\mu$ CSS manual, but in the new syntax.

### 3.1 Named properties (variables)

The simplest use case: a value is named once in the skin manifest and used in any number of places. In the manifest:

```
// skins/src/std.μcss.mjs
import { DefineSkin } from "gulp-mu-css";

export default DefineSkin({
  vars: {
    textColor: "#ff0000",
    backColor: "#0000ff"
  },
  files: [{ source: "src.μ.css", target: "std.css" }]
});
```

In the source stylesheet `src.μ.css`:

```
div.mydiv {
  padding: 5px;
  font-size: 24px;
  color: μ($.textColor);
  background-color: μ($.backColor);
  border: 10px μ($.backColor) solid;
  border-radius: 10px;
}
```

Compiled to `std.css`:

```
div.mydiv {
  padding: 5px;
  font-size: 24px;
  color: #ff0000;
  background-color: #0000ff;
  border: 10px #0000ff solid;
  border-radius: 10px;
}
```

Unlike the old  $\mu$ CSS, no placeholder properties and no `Set*` directives are needed any more: the value sits exactly where it belongs.

### 3.2 Calculations and expressions

The content of  $\mu(\dots)$  is an arbitrary JavaScript expression. This allows calculations, conditions and color operations directly in the stylesheet:

```
td.subheadline {
  border-bottom: 1px dashed μ(Lighten($.selectBaseBrgdColor, 0.7));
}
div.panel {
  padding: μ(Math.floor($.basePadding * 0.2))px;
  background-color: μ(Alpha($.baseBrgdColor, 0.85));
  z-index: μ($.PanelZIndex + 10);
}
div.hint {
  color: μ($.darkMode ? "#e0e0e0" : "#202020");
}
```

For operations that produce several properties or change the rule as a whole, there are directives (`-μ:`), for example using your own macro functions from the manifest:

```
div.menu {
  -μ: Borders($.menuBaseBrngdColor, 1, 0.3, -0.3, -0.3, 0.3);
}
```

The directive calls the helper function `Borders`, which computes four `border-*` properties and inserts them into the rule (see the chapter "`μ` evaluation context").

### 3.3 Multiple layouts via skin manifests

The old template compiling (one template CSS plus a `μ.layout.css` per layout) is replaced by several manifests that share the same `μ.css` sources:

```
// skins/src/layout_a.mjs
import { DefineSkin } from "gulp-mu-css";
export default DefineSkin({
  vars: { textColor: "#ff0000", backColor: "#0000ff" },
  files: [{ source: "template.μ.css", target: "layout_a.css" }]
});

// skins/src/layout_b.mjs
import { DefineSkin } from "gulp-mu-css";
export default DefineSkin({
  vars: { textColor: "#ff00ff", backColor: "#00ff00" },
  files: [{ source: "template.μ.css", target: "layout_b.css" }]
});
```

Each manifest produces its own output directory (`skins/layout_a/`, `skins/layout_b/`) with its own CSS, its own images and its own build cache. Shared variables can be imported as a regular JavaScript module and used in both manifests.

### 3.4 Automatic sprite generation

One of the highlights of `μCSS` (in version 1 as in version 2) is the automatic generation of sprite atlases. The directive `Sprite(url)` registers a rule's image for the atlas:

```
div.loginbutton {
  display: inline-block;
  -μ: Sprite("imgs/aqua/but_login_normal.png");
}
div.loginbutton:hover {
  display: inline-block;
  -μ: Sprite("imgs/aqua/but_login_hover.png");
}
div.helpbutton {
  display: inline-block;
  -μ: Sprite("imgs/aqua/but_help_normal.png");
}
```

During the build, all registered images are packed into a single atlas image (`imgs/sprites.png`, plus `imgs/sprites@2x.png` from the `@2x` source images) and the rules are rewritten:

```
div.loginbutton {
  display: inline-block;
  background-image: url(imgs/sprites.png);
  background-image: image-set(url(imgs/sprites.png)1x, url(imgs/sprites@2x.png)2x);
  background-repeat: no-repeat;
  background-position: 0px 0px;
  width: 55px;
  height: 55px;
}
```

`width`, `height`, `background-position` and `background-repeat` are set automatically; existing declarations of these properties in the rule are replaced. Unlike the old `μCSS`, the vendor-prefixed variants (`-webkit-image-set` etc.) are gone — all relevant browsers have supported `image-set()` unprefixed for years.

Identical source images automatically share one atlas position (deduplication). The atlas is only repacked when the set of images or a source image has changed.

### 3.5 Preloading images (preload)

Important images (e.g. hover states and cursors) can be loaded ahead of time when the page loads. `μCSS` collects the image URLs and generates a preload rule when the manifest option `sprites.preloadRule` is set:

```
div.csspreload {
  background-image: url(imgs/sprites.png), url(imgs/general/gui/cursors/zoom.png);
  display: none;
}
```

In the HTML page, an empty element with this class is enough:

```
<div class="csspreload"></div>
```

Cursor images from the `cursors` definitions are added to the preload list automatically.

### 3.6 Cursors with hotspot and fallback

Cursors are defined in the manifest and used by name in the stylesheet. Definition:

```
cursors: [
  { name: "zoom", fallback: "zoom-in", image: "imgs/general/gui/cursors/zoom.png",
    hotspot: [10, 8] },
  { name: "wait", fallback: "wait", image: "imgs/general/gui/cursors/wait.png", hotspot:
    [12, 12] }
]
```

Usage as a directive or as a value form:

```
*.cursor_zoom {
  -μ: Cursor("zoom");
}
a.preview {
  cursor: μ(Cursor("zoom"));
}
```

The directive produces the `url()` form with hotspot and fallback and — when an `@2x` image file exists — additionally the `image-set()` variant:

```
*.cursor_zoom {
  cursor: url(imgs/general/gui/cursors/zoom.png) 10 8, zoom-in;
  cursor: image-set(url(imgs/general/gui/cursors/zoom.png)1x,
url(imgs/general/gui/cursors/zoom@2x.png)2x) 10 8, zoom-in;
}
```

### 3.7 Automatic image generation (media steps)

What the `μCSS` plugins used to do (`ButtonCreator`, `AppIconMaker`, `FileCopy`) is now handled by the media entries in the manifest. They run before CSS compilation and generate or copy all media into the skin directory:

```
media: [
  { buttonsAndIcons: "dev/media/final/general/gui/panelbuttons.psd", layout: "std",
outputDir: "imgs/general/gui/panelbuttons" },
]
```

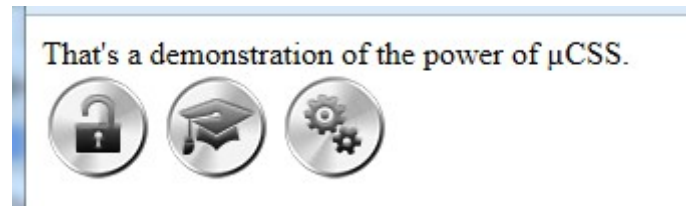


```

{ buttonsAndIcons: "dev/media/final/general/gui/cursors.psd", layout: "std", outputDir:
"imgs/general/gui/cursors" },
{ appIcons: "dev/media/final/favicon.psd", layout: "std", profiles: ["web"] },
{ sequenceStrip: "dev/media/raw/glittery/imgs", outputFile:
"imgs/general/gui/glittery/glittery.png" },
{ copy: "dev/media/final/general/gui/teaserbgd.png", to: "imgs/general/gui" },
{ copyFolder: "dev/media/final/fonts", to: "fonts" }
]

```

- **buttonsAndIcons** renders complete button and icon series including @2x variants from a layered PSD draft (layouts × icon glyphs) — the successor of the ButtonCreator plugin. The layer effects (drop shadow, gradients, glow, etc.) are reproduced by microPS.



*The same icon glyphs, rendered with two layouts of the same draft document ("alu" and "aqua")*



With `mode: "topLayerSets"` the step works in the second legacy mode (`CreateByTopLayerSets`): instead of the `iconxstate` matrix, **each direct subgroup of the layout group becomes exactly one image**, named after the group. No `icons` group is needed here — typical for logos, animation frames (e.g. `activityindicator`) and emoticons. Optionally `setPattern` (regex string) narrows the exported groups.

```

{ buttonsAndIcons: "dev/media/final/general/activityindicator.psd", layout: "std",
mode: "topLayerSets", outputDir: "imgs/general" }

```

- **appIcons** generates app icons and favicons profile-based (`web`, `ios`, `play`) from a square master — the modernized `AppIconMaker`.
- **sequenceStrip** builds a horizontal animation strip with JSON frame data from an image sequence (individual files or one image in DSD format):



*Animation strip generated from 19 individual frames of a smoke animation*

- **copy / copyFolder** copy static files (make-style: only when the source is newer).

The details of the image generators (layer structure of the drafts, DSD format, profiles) are described in the `μPS` documentation.

### 3.7.1 Intermediate steps raw → final (`outputBase``)

By default all steps write into the skin output directory. With `outputBase: "project"` a step writes relative to the project root instead — this lets you express the generation of intermediate results (e.g. sequence strips from `dev/media/raw/...` into `dev/media/final/...`) directly in the manifest. The steps run in manifest order, and a following `copy/copyFolder` step then takes the result into the skin like any other final asset:

```
media: [
```

```
// 1. generate the strip from the raw frames into the final tree ...
{ sequenceStrip: "dev/media/raw/flyex/frames", outputFile:
"dev/media/final/general/gui/flyex/flyex.png", outputBase: "project" },
// 2. ... and copy it from there into the skin as usual.
{ copyFolder: "dev/media/final/general/gui/flyex", to: "imgs/general/gui/flyex",
filter: "\\.(png|json)$" }
]
```

The incremental build applies to these steps too: generation only happens when the result is missing or the configuration or sources (mtime/size) have changed. Alternatively, the raw → final step can of course still run as its own Gulp task before the μCSS build — `outputBase: "project"` is the way to go when everything should live in a single manifest file.

## 4 Core ideas of μCSS

### 4.1 The .μ.css format

Source stylesheets are named \*.μ.css. The double suffix makes editors recognize the files as regular CSS, so syntax highlighting works without further configuration. The extension is irrelevant to the compiler — the manifest references the sources explicitly; anyone who wants to avoid the μ character in file names can use \*.mu.css instead. Content-wise, a .μ.css file is standard CSS with exactly two extension points — both are syntactically valid CSS, so editors and linters work normally:

**1. Value interpolation** μ(expression) — wherever a CSS value can appear.

**2. Directives** -μ: expression; — as a declaration inside a rule.

Anyone without the μ character on their keyboard uses the ASCII aliases mu(...) and -mu: — both forms are equivalent.

Unlike the old μCSS, there are **no control rules** (::-μcss-init etc.) in the stylesheet any more: everything that controls compilation (variables, cursor definitions, media generation, file mapping) lives in the skin manifest — a regular JavaScript file (see below).

### 4.2 Value interpolation μ(expression)

From a CSS point of view, μ(...) is an unknown but valid function. During compilation its content is evaluated as a JavaScript expression and the μ(...) occurrence is replaced by the result:

```
::selection {
  background-color: μ($.selectBaseBrkdOnDarkColor);
  color: μ($.selectBaseTextColor);
}
```

Multiple interpolations per value are allowed (padding: μ(\$.padY)px μ(\$.padX)px;), as are interpolations in at-rule parameters (e.g. @media (max-width: μ(\$.breakpoint)px)). If an expression returns null or undefined, compilation aborts with an error message — so typos in variable names show up immediately.

This single extension replaces the entire Set\* family of the old μCSS (SetColor, SetBackgroundColor, SetZIndex, SetWidth, AddProperty for simple values, etc.): the property is back in its place, and a placeholder value is no longer needed.

### 4.3 Directives -μ: expression

Directives are declarations with the property name -μ (or -mu). Their value is a single JavaScript expression executed with access to the surrounding rule and the document. The directive itself is removed from the output:

```
div.panel.modal div.companylogo {
  -μ: Sprite("imgs/logos/dosing_logo.png");
  margin-left: auto;
}
*.cursor_zoom {
  -μ: Cursor("zoom");
}
div.content.glittery {
  -μ: Sprite("imgs/general/gui/glittery/glittery.png", { afterWork: GlitterySprite });
}
```

The μ. prefix of the old μCSS is gone — the context is implicit. Directives are evaluated in document order.

## 4.4 The skin manifest

Per skin (CSS theme) there is a manifest file `<skinname>.μcss.mjs` in the source directory — regular JavaScript (ES6+), importable and testable. The double suffix `.μcss.mjs` (ASCII alternative `.mucss.mjs`) clearly marks the file as a μCSS skin manifest so it is not confused with an ordinary module or gulpfile; the skin name is the part before it (`std.μcss.mjs` → skin `std`). It fully replaces the old control file `μ.std.css`:

```
// skins/src/std.μcss.mjs – skin "std", target: skins/std/
import { DefineSkin } from "gulp-mu-css";
import { GlitterySprite, FlyEx, Borders, TableBackgrounds } from "./helpers.mjs";

export default DefineSkin({
  vars: {
    baseBrgdColor: "#202020",
    selectBaseBrgdColor: "#007570",
    PanelZIndex: 9000
  },
  helpers: { GlitterySprite, FlyEx, Borders, TableBackgrounds },
  cursors: [
    { name: "zoom", fallback: "zoom-in", image: "imgs/general/gui/cursors/zoom.png",
      hotspot: [10, 8] }
  ],
  media: [
    { buttonsAndIcons: "dev/media/final/general/gui/panelbuttons.psd", layout: "std",
      outputDir: "imgs/general/gui/panelbuttons" },
    { copyFolder: "dev/media/final/fonts", to: "fonts" }
  ],
  imageFormat: "png",
  sprites: { file: "imgs/sprites.png", retina: true, preloadRule: true },
  files: [
    { source: "src.μ.css", target: "std.css" },
    { source: "src_tinymce.μ.css", target: "std_tinymce.css" }
  ]
});
```

The skin name is derived from the manifest file name, the output directory from the skin name. The complete field reference is in the chapter "Manifest reference".

## 4.5 JavaScript without substitute characters

In μCSS 1, the characters `{`, `}` and `;` had to be replaced by `«`, `»` and `¡` in JavaScript statements to avoid CSS syntax conflicts. This pattern is gone in version 2:

- `μ(...)` and `-μ: ...` contain **single expressions** — the parser counts parentheses and accounts for strings, so object literals (`{ afterWork: ... }`) and nested calls work without trouble.
- **Multi-line logic** (loops, function definitions) does not belong in the stylesheet but in the manifest's helper modules — real `.mjs` files with syntax highlighting, linting and a debugger.

## 5 Build process and Gulp integration

The old `μCSS` was operated through a modal dialog window in Photoshop. `BuildSkin` and Gulp take its place: the build is a regular, scriptable Node process — suitable for watch mode, CI and automation.

### 5.1 Directory conventions

For a manifest `skins/src/std.μcss.mjs` the following applies (all paths overridable):

Path	Meaning
<code>skins/src/</code>	Source directory: manifests, <code>.μ.css</code> files, helper modules
<code>skins/std/</code>	Output directory of the "std" skin: CSS, images, fonts, sounds
<code>skins/std/.cache/build.json</code>	Build cache of the skin (fingerprints, atlas positions)
Project root	Two levels above the manifest; base for <code>media</code> source paths like <code>dev/media/...</code>

`BuildSkin(manifestPath, options)` accepts `{ outputDir, rootDir, force }` for overriding.

### 5.2 Compilation flow

A `BuildSkin` run performs five steps:

- 1. media steps:** all images are generated or copied (microPS) — they must exist before the atlas and cursor checks run.
- 2. Compilation:** each `files` source is parsed (PostCSS); directives and interpolations are evaluated in document order. `Sprite()/Cursor()` calls initially only register their image references.
- 3. Sprite atlas:** all registered images are packed (incl. `@2x`); afterwards the affected rules are rewritten.
- 4. Hooks:** `afterWork` callbacks run with the atlas result and AST access.
- 5. Output:** the compiled CSS files are written into the skin directory, the cache is saved.

### 5.3 Incremental build and cache

Each run only regenerates what actually changed. The primary mechanism is file modification timestamps (`mtime` plus file size) — orders of magnitude faster than content hashes for large PSD sources:

- **Generator steps** (`buttonsAndIcons`, `appIcons`, `sequenceStrip`) are skipped when their configuration is unchanged, all source files have unchanged fingerprints and all output files still exist. PSD rendering is the most expensive part of the build — this is where the cache pays off the most.
- `copy/copyFolder` work make-style: copying happens only when the source is newer than the target.
- **The sprite atlas** is only repacked when the set of images or a source image (incl. `@2x`) has changed — otherwise the stored positions are reused and only the CSS rules are rewritten with them.
- **CSS compilation** itself is cheap and always runs when in doubt.

The cache lives per skin in `<outputDir>/cache/build.json` and carries the version numbers of `μCSS` and the cache schema; on a mismatch it is discarded (full build). `BuildSkin(manifest, { force: true })` or deleting `cache/` forces a full build explicitly.

## 5.4 Gulp tasks and watch

One task per skin is enough; `BuildSkin` is re-entrant:

```
import gulp from "gulp";
import { BuildSkin } from "gulp-mu-css";

export async function SkinStd() {
  const report = await BuildSkin("skins/src/std.μcss.mjs");
  console.log(`${report.skin}: ${report.files.length} files, atlas $
{report.atlasSkipped ? "from cache" : "repacked"}, ${report.duration} ms`);
}
export function SkinWatch() {
  gulp.watch(["skins/src/**/*.μ.css", "skins/src/*.mjs", "dev/media/final/**"], SkinStd);
}
```

The return value of `BuildSkin` is a report with `skin`, `outputDir`, `media` (step results with a `skipped` flag), `files`, `atlas`, `atlasSkipped` and `duration`.

## 6 Manifest reference (DefineSkin)

`DefineSkin(configuration)` declares the skin configuration (the manifest's default export) and validates the basic structure. All fields are optional unless stated otherwise.

### 6.1 vars

Object with the skin variables — the replacement for `μ.$.*`. Accessible in `μ.css` expressions as `$.name`, in helper functions as `this.$.name`. Values are arbitrary JavaScript values (strings, numbers, objects, even computed ones).

```
vars: { baseBrgdColor: "#202020", PanelZIndex: 9000, icon_pencil: "\\e91c" }
```

### 6.2 helpers

Object with user-defined functions (macros and hooks). They are available in `μ.css` expressions under their name. Functions declared with `function` (not as an arrow function) receive `this` = the evaluation scope when called (see the chapter "μ evaluation context").

```
import { Borders, GlitterySprite } from "./helpers.mjs";
// ...
helpers: { Borders, GlitterySprite }
```

### 6.3 cursors

List of cursor definitions — the replacement for `μ.DefCursor`:

Field	Default	Meaning
name	— (required)	Name under which the cursor is used via <code>Cursor("name")</code> .
fallback	= name	Standard cursor name (W3C) as a fallback.
image	""	Image URL relative to the skin directory; empty = definition only as a fallback mapping.
hotspot	[0, 0]	Click point [x, y] in the image; 0,0 is omitted in the output.
forceFallback	false	Appends the fallback additionally as the last cursor declaration.

Cursor images are added to the preload list automatically.

### 6.4 media

List of media steps; they run in the given order before compilation. The step type is determined by the key field:

Step	Required fields	Further fields	Effect
buttonsAndIcons	source PSD, layout, outputDir	retina (default true), format, mode, setPattern	Render button/icon series from a draft document (microPS ButtonAndIconCreator) . mode: "topLayerSets"

Step	Required fields	Further fields	Effect
			switches to "one image per top subgroup" (legacy <code>CreateByTopLayerSets</code> , no icons group — for logos, animation frames, emoticons); <code>setPattern</code> (regex string) filters the exported groups.
<code>appIcons</code>	source PSD/PNG	<code>outputDir</code> , <code>profiles</code> , <code>layout</code> , <code>background</code> , <code>appName</code> , <code>shortName</code> , <code>themeColor</code>	Generate app icons/favicons profile-based (microPS <code>AppIconMaker</code> ).
<code>sequenceStrip</code>	source (folder or DSD image), <code>outputFile</code>	<code>retina</code> (default <code>true</code> ), <code>writeMapFile</code> (default <code>true</code> ), <code>format</code>	Generate a horizontal animation strip plus JSON frame data (microPS <code>SequenceStrip</code> ).
<code>copy</code>	source file	<code>to</code> (target folder, default skin root)	Copy a single file when the source is newer.
<code>copyFolder</code>	source folder	<code>to</code> (default = folder name), <code>filter</code> (regex string, e.g. "\\.(woff2? ttf)\$")	Copy a folder recursively (make-style), optionally filtered.

Source paths are relative to the project root, target paths relative to the skin directory. Each step additionally accepts `outputBase`: "skin" (default) or "project": with "project" the target path is relative to the project root — for intermediate steps like raw → final (see the chapter "Automatic image generation").

## 6.5 imageFormat

Global image format switch: "png" (default) or "webp". **Scope:**

- **Image-generating media steps** (`buttonsAndIcons`, `appIcons*`, `sequenceStrip`) and the **sprite atlas** (unless `sprites.format` is set, see below).
- Individual steps can override the format via their own `format` field.
- Directly referenced existing images (`url(...)` in the sources, `copy/copyFolder` steps) are left untouched — a `copyFolder filter` only copies what it lets through. If the `filter` excludes the active `imageFormat` extension (e.g. "\\.(png|json)\$" with `imageFormat`: "webp"), the build emits a **warning** instead of silently discarding the freshly generated files.

\*`\*appIcons` produces platform-specific icon sets (PNG, partly ICO) and **deliberately ignores** `imageFormat` — app/favicon formats are fixed.\*

## 6.6 sprites

Options of the sprite atlas — the replacement for `μ.options.sprites.*`:



Field	Default	Meaning
file	"imgs/sprites.png"	Atlas file (URL as it appears in the CSS).
format	(none)	Atlas output format ("png"/"webp"), <b>independent of imageFormat</b> . When set, it overrides the extension derived from imageFormat; otherwise imageFormat applies. This lets you switch the atlas alone to WebP (sprites: { file: "imgs/sprites.png", format: "webp" }) without touching the generator steps — the Sprite() source images may stay PNG.
retina	true	Additionally generates <name>@2x from the @2x source images (which must sit next to the 1x sources).
padding	0	Spacing in pixels between the sprites.
preloadRule	false	Generates the div.csspreload rule in the first stylesheet.

**Resolution of the Sprite() source images:** the path referenced in Sprite("...") is resolved format-independently — first the literal path, then the same name with one of the supported raster extensions (png, webp). So if a generated source image exists as PNG even though the CSS reference says .webp (or vice versa), the build does not abort. An error only occurs when **no** variant exists. The @2x variant must have the same extension as the resolved 1x image.

## 6.7 files

List of stylesheets to compile — the replacement for `μ.DependentCSSFile`. `source` is relative to the source directory, `target` relative to the skin directory; both fields are required:

```
files: [
  { source: "src.μ.css", target: "std.css" },
  { source: "src_tinymce.μ.css", target: "std_tinymce.css" }
]
```

All files of a skin share variables, helpers and the sprite atlas. The preload rule goes into the first stylesheet.

## 7 $\mu$ evaluation context (reference)

Every  $\mu(\dots)$  expression and every  $-\mu:$  directive is evaluated in a scope that contains the following bindings. It corresponds to the old global  $\mu$  object — just without the prefix.

### 7.1 The \$ object

\$ contains the manifest's vars. Reading and writing are possible; changes apply for the rest of compilation (document order, across all files of a skin).

```
div.box { z-index:  $\mu(\$.PanelZIndex + 1)$ ; }
```

### 7.2 Color and value functions

Function	Description
Lighten(color, step, model = "hsl")	Lightens a color relatively (positive step) or darkens it (negative step): $L' = \text{clamp}(L + L \cdot \text{step})$ . The model "hsl" is bit-identical to the old $\mu$ CSS; "oklch" scales perceptually uniformly. Alpha is preserved.
Alpha(color, alpha)	Replaces a color's alpha channel. alpha follows the rules from "Working with colors".
MixColors(color1, color2)	Per-channel average of two colors (including alpha).
AlphaValue(alpha)	Converts an alpha specification to a byte (0–255).
ParseColor(color)	Parses a CSS color into the internal 32-bit representation 0xAARRGGBB.
FormatColor(color, alphaDecimals = 3)	Serializes the internal representation back to CSS (#rrggbb or rgba(...)).
PxUnit(value)	Numbers become "<n>px", everything else (e.g. calc() strings) is passed through unchanged.

### 7.3 Rule and document methods

Inside directives (and helper functions with a this binding) the manipulation methods of the surrounding rule are additionally available:

Method / property	Description
AddProperty(name, value, important?)	Appends a declaration to the rule (duplicates allowed — for fallback chains like multiple background-image).
ChangeProperty(name, value, important?)	Changes all declarations of the property, or creates one if it is missing.
RemoveProperty(name)	Removes all declarations of the property.
AddRule(selector)	Appends a new rule at the end of the document; returns the rule (with the same methods).
InsertRule(selector)	Inserts a new rule directly after the current rule; consecutive calls keep their order. Replacement

Method / property	Description
	for the old <code>AddBlock(n, μ.elementNo)</code> .
<code>rule</code>	The surrounding rule as a <code>CssRule</code> object ( <code>rule.selector</code> , <code>rule.GetProperty(...)</code> , ...).
<code>document</code>	The entire stylesheet as a <code>CssDocument</code> — e.g. for path addressing: <code>document.FindRule("@keyframes glittery", "from")</code> . Replacement for the old <code>RememberBlocks</code> .

## 7.4 Sprite()

Registers the rule's image for the sprite atlas (directive only):

```
-μ: Sprite(url, options?);
```

Parameter / option	Default	Description
<code>url</code>	— (required)	Image URL relative to the skin directory.
<code>offsetWidth</code>	0	Added to the computed width.
<code>offsetHeight</code>	0	Added to the computed height.
<code>offsetPosX</code>	0	Added to the background-position X coordinate.
<code>offsetPosY</code>	0	Added to the background-position Y coordinate.
<code>afterWork</code>	—	Hook function, runs after atlas resolution (see below).

During atlas resolution the rule's `background-image` (`url(...)` plus `image-set(...)` with retina), `background-repeat`, `background-position`, `width` and `height` are set; existing declarations of these properties are replaced.

## 7.5 Cursor()

Applies a cursor definition from the manifest — as a directive (`-μ: Cursor("zoom");`, rewrites the rule's `cursor` declarations) or as a value form (`cursor: μ(Cursor("zoom"));`, returns only the `url(...)` x y, fallback string). Unknown names are passed through unchanged — standard cursors like `pointer` thus work without a definition.

## 7.6 Helper functions and the this binding

Helpers from the manifest declared with `function` are called with `this` = the evaluation scope. This lets you write macros in the style of the old `μ.$.` functions — just as regular, testable JavaScript:

```
// helpers.mjs
import { Lighten } from "gulp-mu-css";

export function Borders(_baseColor, _pixelWidth, _topLighten, _rightLighten,
  _bottomLighten, _leftLighten) {
  this.AddProperty("border-top", `${_pixelWidth}px solid ${Lighten(_baseColor,
    _topLighten)}`);
}
```

```

    this.AddProperty("border-right", `${_pixelWidth}px solid ${Lighten(_baseColor,
_rightLighten)}`);
    this.AddProperty("border-bottom", `${_pixelWidth}px solid ${Lighten(_baseColor,
_bottomLighten)}`);
    this.AddProperty("border-left", `${_pixelWidth}px solid ${Lighten(_baseColor,
_leftLighten)}`);
}

```

Through this all scope bindings are reachable: `this.AddProperty(...)`, `this.InsertRule(...)`, `this.rule`, `this.document` and `this.$`. The binding is also kept when a helper is passed as an `afterWork` value to `Sprite()`. Arrow functions have no `this` of their own and are therefore only suitable for helpers without rule access.

Ported reference macros of the AiDPix project (`Borders`, `TableBackgrounds`, `GlitterySprite`, `FlyEx`, `FlyExUtils`) live in the `μCSS` repository under `test/fixtures/aidpix-helpers.mjs`.

## 7.7 afterWork hooks

The `afterWork` hook of a sprite registration runs after the atlas has been packed and the rule rewritten. It receives a context with all result data:

Field	Description
<code>rule</code>	The rewritten rule ( <code>CssRule</code> ).
<code>document</code>	The stylesheet ( <code>CssDocument</code> ).
<code>url</code>	The registered image URL.
<code>baseDir</code>	Skin directory (for resolving neighboring files, e.g. <code>.json</code> maps).
<code>sprite</code>	<code>{ x, y, width, height }</code> — position and size in the atlas.
<code>atlas</code>	<code>{ file, retinaFile, width, height }</code> — the atlas files and total size.

Typical use: generating animation keyframes from the frame data of a `sequenceStrip` (see `GlitterySprite` in the reference macros).

## 8 Working with colors

### 8.1 Defining colors

The internal representation of a color is an unsigned 32-bit integer of the form `0xAARRGGBB` (alpha, red, green, blue; 8 bits each). All color functions accept the following input forms:

- **32-bit integer** like the internal representation, e.g. `0xffff0000` for opaque red.
- **# notation** with two hex digits per channel, e.g. `"#00ff00"` (alpha is set to opaque).
- **Short # notation** with one hex digit per channel, e.g. `"#0f0"`.
- **Extended # notation** with an alpha channel as the fourth component, e.g. `"#0000ff80"` for semi-transparent blue.
- **rgb() notation**, absolute or percentage, e.g. `"rgb(255,0,0)"` or `"rgb(100%,0%,0%)"`.
- **rgba() notation** with float alpha, e.g. `"rgba(255,0,0,0.5)"`.
- **CSS color names** (W3C CSS Color Module, extended list), e.g. `"red"`, `"teal"`, `"rebeccapurple"` as well as `"transparent"`.

Values outside the valid range are clipped.

### 8.2 Alpha values

Alpha specifications (e.g. for `Alpha(color, a)`) are possible in several forms:

- **Float** between `0.0` (transparent) and `1.0` (opaque).
- **Integer** between `0` and `255`.
- **Hex string**, e.g. `"0x80"`.
- **Percent string** from `"0%"` to `"100%"`.
- **Keywords** `"transparent"` (0), `"translucent"` (128) and `"opaque"` (255).

### 8.3 Color computations

The functions `Lighten`, `Alpha` and `MixColors` (chapter "μ evaluation context") cover the typical computations. Opaque results are emitted as `#rrggbb`, transparent ones as `rgba(r,g,b,a)` with three alpha decimals — identical to the old `μCSS`.

```
div.menu {
  background-color: μ(Alpha($.baseBrgdColor, 0.9));
  border-top: 1px solid μ(Lighten($.baseBrgdColor, 0.3));
  border-bottom: 1px solid μ(Lighten($.baseBrgdColor, -0.3));
}
```

For new skins, the `"oklch"` model of `Lighten` is recommended: it changes the perceived lightness uniformly across all hues, whereas the legacy `"hsl"` model can jump visibly for saturated colors.

## 9 Node API

Besides the manifest workflow, every layer of  $\mu$ CSS is also usable directly as a Node API — for example for your own Gulp transforms or tests.

Export	Description
<code>CompileMcCss(source, options)</code>	Compiles <code>.<math>\mu</math>.css</code> source text into a <code>CssDocument</code> . Options: <code>vars</code> , <code>helpers</code> , <code>from</code> (file name for error messages), <code>context</code> (shared <code>MuContext</code> ), <code>sprites</code> ( <code>SpriteManager</code> ), <code>cursors</code> ( <code>CursorManager</code> ).
<code>CssDocument</code> / <code>CssRule</code>	JSON-capable wrapper around the PostCSS AST: <code>FromFile/FromString</code> , <code>FindRule(...path)</code> , <code>FindRules(selectorOrRegex)</code> , <code>AddRule</code> , <code>GetProperty/AddProperty/ChangeProperty/RemoveProperty</code> , <code>ToCss()</code> , <code>ToFile()</code> , <code>ToJson()/FromJson()</code> . For special cases the raw PostCSS AST is accessible via <code>document.root</code> .
<code>SpriteManager</code>	Sprite registration and atlas resolution: <code>new SpriteManager({ baseDir, atlasFile, retina, padding, preloadRule, preload, writeMapFile })</code> , <code>Register(rule, url, options)</code> , <code>await Resolve(document)</code> .
<code>CursorManager</code>	Cursor definitions: <code>new CursorManager(definitions, { baseDir, preload })</code> , <code>Apply(rule, name)</code> , <code>Value(name)</code> .
<code>PreloadRegistry</code>	Collects image URLs and generates the <code>div.csspreload</code> rule.
<code>DefineSkin(config)</code> / <code>BuildSkin(manifest, options)</code>	Manifest declaration and skin build (see the chapter "Build process").
<code>MuContext</code>	Evaluation context for your own pipelines: <code>new MuContext({ vars, helpers })</code> , <code>Evaluate(sourceText, extraScope)</code> .
<code>Lighten</code> , <code>Alpha</code> , <code>AlphaValue</code> , <code>MixColors</code> , <code>ParseColor</code> , <code>FormatColor</code> , <code>PxUnit</code>	The color and value functions as direct imports.

Example of a JSON-based Gulp manipulation:

```
import { CssDocument } from "gulp-mu-css";

const doc = await CssDocument.FromFile("skins/src/src. $\mu$ .css");
doc.FindRules(/^div\.panel/).forEach((_rule) => _rule.AddProperty("outline", "none"));
doc.FindRule("@keyframes glittery", "from").ChangeProperty("background-position-x", "-128px");
const json = doc.ToJson();
await doc.ToFile("out/std.css");
```

## 10 Error diagnostics

Build errors always name the culprit together with its source location:

- **Inline JavaScript** ( $\mu(\dots)$  interpolations and  $-\mu$ : directives): errors are reported as PostCSS errors with file, line, column and a source excerpt. With multiple  $\mu(\dots)$  in one value, the failing expression is named. JavaScript syntax errors appear as `invalid JavaScript expression "<source>" (...)`.

CssSyntaxError: skins/src/std. $\mu$ .css:2:2: microCSS:  $\mu$ (NopeFn(1)): NopeFn is not defined

```
1 | div.b {  
> 2 |     border: 1px solid  $\mu$ (NopeFn(1));  
   |           ^  
3 | }
```

- **Missing sprite images**: before the atlas is packed, the existence of all source images (including @2x with `retina: true`) is checked. All missing files are reported together in one error — each with URL, resolved path and the referencing rule:

SpriteManager: 2 sprite image(s) not found:

```
- "imgs/nope.png" -> C:\...\skins\std\imgs\nope.png - selector "div.bad1" (skin. $\mu$ .css:2)  
- "imgs/alsonope.png" -> C:\...\skins\std\imgs\alsonope.png - selector "div.bad2"  
(skin. $\mu$ .css:3)
```

- **afterWork hooks**: errors in the hook are wrapped with the sprite URL and the rule's source position; the original error is preserved as `cause`.
- **Missing cursor images** produce only a warning (once per cursor), since the CSS stays functional via the fallback cursor.
- **media steps**: errors name the step number and type, e.g. `BuildSkin: media step 3 of 7 (buttonsAndIcons: "dev/media/buttons.psd") failed: ....` Missing sources are checked before execution: `copy/copyFolder` and generator steps report the resolved path instead of a raw file system error — for `copyFolder` with the hint that the generating step (e.g. sequence-image generation into `dev/media/final/...`) probably did not run.
- **files entries**: missing source files are reported with the entry and the resolved path before compilation.

## 11 Migrating from μCSS

For existing projects there is a converter tool (`tools/convert-mucss.mjs` in the μCSS repository) that mechanically translates the old syntax:

Old (μCSS 1)	New (μCSS 2)
<code>-μcss: μ.Cursor("wait"); plus cursor: wait;</code>	<code>cursor: μ(Cursor("wait"));</code>
<code>-μcss: μ.SetBackgroundColor(μ.\$.x); plus placeholder</code>	<code>background-color: μ(\$.x);</code>
<code>-μcss: μ.AddProperty("border", "1px solid " + μ.Lighten(...));</code>	<code>border: 1px solid μ(Lighten(...));</code>
<code>-μcss: μ.Sprite("p.png");</code>	<code>-μ: Sprite("p.png");</code>
<code>-μcss: μ.SetRememberBlock("n");</code>	gone — path addressing: <code>document.FindRule("@keyframes x", "from")</code>
<code>-μcss: //μ.X(...) (disabled)</code>	<code>/* -μ: X(...) */</code>
<code>μ.\$.name = value (in μ.std.css)</code>	vars entry in the manifest
<code>μ.DefCursor(...)</code>	cursors entry in the manifest
<code>μ.plugins.* / FileCopy</code>	media entries in the manifest
<code>μ.\$.Fn = function(...)«...»</code>	exported function in <code>helpers.mjs</code>

The «»i function bodies are translated back to regular JavaScript and land as a starting point in `helpers.mjs`; manual rework is expected here.

**raw → final automatically:** the old μCSS only copied the already finished `final` folders into the skin (e.g. `flyex`, `glittery`); the strips themselves were created elsewhere (`SpriteTools`). The converter recognizes such `CopyFolder2Skin` calls on `dev/media/final/<...>/<name>` and — if a matching source `dev/media/raw/<name>/imgs` exists — automatically prepends the appropriate `sequenceStrip` step with `outputBase: "project": a numbered PNG frame sequence (e.g. glittery) becomes a folder strip, individually named images (e.g. the DSD images flyex.png/flyexutils.png) each become a DSD strip.` This way the new pipeline rebuilds `final` reproducibly from `raw` and the following `copyFolder` takes the result into the skin.

The converter was validated against the complete AiDPix code base: a comparison tool (`tools/compare-aidpix.mjs`) builds the converted skin and compares it rule by rule and property by property against the old compiled output — result: 2,951 rules, 0 unexpected differences (53 documented drift cases, e.g. sources edited after the last μCSS run).

Deliberately **not** carried over from the old μCSS:

- **Vendor prefixes** (`-webkit-/-moz-/-ms-` duplicates): all used features have been available unprefixed for years. Vendor-specific selectors without a standard counterpart (e.g. `::-webkit-scrollbar`) are of course passed through unchanged.
- **The Photoshop dialog window** in the build: interactive parameters belong in the manifest or in environment variables.
- **FTP synchronization:** there are established tools for that today (CI/CD, `rsync`, gulp plugins).
- **The substitute characters** «»i: see the chapter "Core ideas".



## 12 Version history

Date	Version	Notes
2013	1.0	Original <code>μCSS</code> as an Adobe Photoshop script: <code>-μcss:</code> directives, sprite atlas, PSD plugins, control via <code>μ.std.css</code> .
2026-06	2.0.0	Complete Node.js reimplementation (npm package <code>gulp-mu-css</code> , without Adobe dependency): core pipeline (M1), sprites & cursors (M2), manifest & build with incremental cache (M3), hooks & macros (M4), AiDPix migration with converter and acceptance test (M5), manual (M6).
2026-06	2.1.0	Atlas format independent of the global <code>imageFormat</code> via <code>sprites.format</code> ; format-independent resolution of the <code>Sprite()</code> source images ( <code>png/webp</code> ); warning when a <code>copyFolder</code> filter excludes the active image format.
2026-06	2.2.0	Normalization of legacy gradient directions at compile time: <code>linear-gradient(top bottom left right, ...)</code> (invalid without a prefix) is raised to the standard to <code>...</code> form.
2026-06	2.2.1	Manual update (version history extended, cover-page version corrected); no functional changes compared to 2.2.0.
2026-06	2.2.2	Bilingual manual ( <code>microCSS-de.pdf</code> / <code>microCSS-en.pdf</code> ) and bilingual READMEs (English first) for <code>gulp-mu-css</code> and <code>gulp-mu-ps</code> ; build tooling extended for multilingual manuals. No functional code changes.

## 13 Legal

This software is released under the MIT license and is free for both free and commercial use. Neither Dongleware nor the author is liable for any damage arising from the use of this software. Use is at your own risk; back up your work before using the tools.

Author: Meinolf Amekudzi.

### 13.1 MIT license (original text)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 13.2 Trademarks

Photoshop is a registered trademark of Adobe Inc.; Affinity Photo is a trademark of Serif (Europe) Ltd. Names and products are mentioned for information only and do not constitute any misuse of the respective trade names or trademarks.

### 13.3 Third-party libraries

μCSS and μPS use third-party open-source libraries, in particular PostCSS (CSS parser, MIT license), sharp (image processing, Apache 2.0 license) and ag-psd (PSD reader, MIT license). The sprite atlas bin packer is based on node-bin-packing (©2011 Jake Gordon and contributors, MIT license).