Table des matières

1. µcBlockly – Project architecture and contributor guide

This document helps new contributors understand the codebase layout, conventions, and where to change things.

> **Noyau vs démo** : pour l'API publique embed (`createUcBlockly`), les builds sans Monaco et la frontière interne/public, voir **KERNEL.md**.

1.1. Overview

µcBlockly (v3) is a visual programming environment based on **Blockly 13** that generates Arduino/C++ code. Users assemble blocks in the workspace; the Arduino generator turns the block tree into C++. The demo app also includes a Monaco code panel, board profiles, and optional plugins (minimap, backpack, search, content highlight, etc.).

Public surfaces: **`µcBlockly/core`** (embed kernel), **`µcBlockly/host`** (product/shell helpers), **`µcBlockly/demo`** (full demo entry). See KERNEL.md.

1.2. Source layout (''src/'')

```
src/
├── index.ts                    # Demo entry: Blockly global exposure,
workspace init, UI wiring
├── blockly_application_type.ts # Re-export →
demo/blockly_application_shell.ts
├── core/                       # **Public kernel** — createUcBlockly,
workspace manager (see KERNEL.md)
├── host/                       # **Product helpers** — themes, a11y, Monaco
wiring (not in lean kernel)
├── demo/                       # Demo shell + embed example (internal)
├── blockly_patches.ts          # Defensive patches for Blockly (e.g.
removeTopBlock during drag)
├── toolbox.ts                  # Toolbox structure: categories, block
lists, board-specific toolbox
├── toolbox_search_i18n.ts      # i18n wrapper for @blockly/toolbox-search
(Blockly.Msg)
├── options.ts                  # Theme extension (libre/board styles),
themeMappings
├── tools.ts                    # URL/query helpers (addReplaceParamToUrl,
setPluginsInURL)
├── language.ts                 # Language switching, toolbox translation,
URL lang param
├── types.d.ts                  # Ambient declarations (Blockly plugins,
```

```
Monaco, etc.)
├── boards.ts                   # Board profiles (pins, PWM, serial, etc.) —
data + helpers
├── code_editor.ts              # Lazy Monaco editor (generated C++ panel);
shared µcB_codeEditor
├── workspace_resize.ts         # Blockly.svgResize +
registerWorkspaceForSvgResize (no import from index)
├── workspace_flex.ts           # Flex resizers, panel toggles,
saveLayoutToSessionStorage, µcB_addFlexResizerEvents
├── workspace_layout_utils.ts   # Flex helpers (parsePositiveFlex,
snapNearZeroFlex, layout keys)
├── a11y_labels.ts              # getA11yLabel for layout UI (avoids
importing language.ts from flex stack)
├── shell_a11y.ts               # Skip link, menu zones Ctrl+B, landmarks
coque HTML
├── shell_shortcuts_help.ts     # Shortcuts help dialog (F1 / ?)
├── shell_keyboard_tutorial.ts  # First-run keyboard tutorial
├── ui_palette_theme.ts         # syncUiPaletteTheme: coque CSS + Monaco
depuis thème Blockly
│
├── categories/blockly/         # Block *definitions* (UI blocks, fields,
mutators)
│   ├── index.ts                # Entry: defineBlocklyArduinoCompatBlocks,
defineAllTypedVariableBlocks
│   ├── array.ts, array_bitmap*.ts  # Array/list + bitmap blocks
│   ├── board.ts                # Board blocks (pins, serial, etc.)
│   ├── colour.ts               # Colour blocks
│   ├── libre.ts                # “Libre” / free-text blocks (ex
Blockly@rduino null category)
│   ├── logic.ts, loops.ts      # Logic/compare + loops
│   ├── math.ts, text.ts
│   ├── variables.ts            # Typed variables (get/set/define/local),
extensions
│   ├── shared.ts, shared_numeric.ts
│   └── field_*.ts              # Custom fields (colour, slider, bitmap,
HSV)
│
├── generators/arduino/         # Arduino *code generation* (block → C++
string)
│   ├── arduino_generator.ts    # Main generator class (variables,
functions, loops, finish)
│   ├── block_types.ts          # Type compatibility
(int/float/bool/string/…), TYPE_COMPATIBILITY
│   ├── block_types_registry.ts # Applies block types to Blockly (setCheck),
BLOCK_TYPE_REGISTRY
│   ├── block_types/            # Per-category type definitions (static +
__DYNAMIC__)
│   ├── blocks/                 # Per-block code generators (one file per
category)
│   ├── configure_standard_blocks.ts # Standard Blockly blocks type checks
│   ├── strict_connection_checker.ts # Optional strict connection checker
```

```
│   ├── strict_type_enforcer.ts # Type enforcer (e.g. variable types)
│   ├── typed_variables_flyout.ts   # Typed variables category in toolbox
│   ├── variable_utils.ts       # Variable type normalization
│   ├── arduino_types.ts        # Arduino type metadata (default values,
etc.)
│   └── debug.ts                # debugLog (no-op unless enabled)
├── generators/arduino.ts       # Entry: arduinoGenerator, registers all
block generators
│
└── languages/                  # UI strings (Blockly.Msg, toolbox labels)
    ├── languageMap.ts          # Language list and mapping
    ├── en.ts, fr.ts, es.ts, ar.ts
    ├── array_messages.ts       # Array block messages (mergeArrayMessages)
    └── board_messages.ts       # Board block messages (mergeBoardMessages)
```

Hors `src/` :

```
scripts/                        # Outils CLI (maintenance, conversion,
assets)
├── check-translations.cjs      # Cohérence des clés i18n
├── convert-rduino-workspace.mjs # Blockly@rduino XML → JSON µcBlockly
├── generate-favicon.mjs        # Favicon depuis logo_µcBlockly.png
├── extract-functions.js        # Régénère docs/functions-list.md
└── lib/                        # Modules partagés (rduino-block-map,
rduino-xml-to-ucb-json)

tests/                          # Tests Node (tsx --test)
├── generators.test.ts          # Génération Arduino (switch, listes,
setup/loop…)
├── logic-compare-connections.test.ts
├── math-change-typing.test.ts
├── list-declaration-uniqueness.test.ts
├── core-api.test.ts            # API noyau / createUcBlockly
├── rduino-convert.test.ts      # Convertisseur Blockly@rduino
├── array-bitmap.test.ts
└── field-colour-a11y.test.ts
```

Voir CONVERT_RDUINO.md pour la conversion de projets Blockly@rduino.

1.3. Conventions

- **Comments (TSDoc / TypeDoc)**: In English. Use `@description`, `@remarks`, `@param`, `@returns` for public APIs. File-level `@packageDocumentation` (with optional `@module`) describes the module for the generated API site. Avoid legacy **`@fileoverview`** (JSDoc); fold that text into `@remarks` or `@description`.
- **License header**: Each source file starts with `@packageDocumentation` where applicable, then the GPL-3.0-or-later license block. Author/copyright (e.g. ASTUCE) is kept as-is.
- **Block definitions vs code generation**:
- **categories/blockly/** = what the user sees (block shape, fields, connections). Register with

`Blockly.Blocks["block_type"] = { init() { … } }` or `Blockly.defineBlocksWithJsonArray`.
- **generators/arduino/blocks/** = how each block becomes C++ (e.g. `arduinoGenerator.forBlock["block_type"] = function(block, generator) { return "code"; }`).
- **Block types**: Typing (int/float/bool/string/char/long/…) is defined in **block_types/** and **block_types.ts** (TYPE_COMPATIBILITY, cast mapping). **block_types_registry.ts** applies types and runs `checkTypeCompatibility`. See **TYPES_ET_COMPATIBILITES.md** for the full matrix and cast rules.
- **Board-specific behaviour**: **boards.ts** holds board profiles (pins, PWM, serial, etc.). **toolbox.ts** uses `getToolboxForCurrentBoard()` to build the toolbox for the selected board. Pin dropdowns and validators are wired from **categories/blockly/board.ts** and refreshed via `refreshBoardPinDropdowns`.

1.4. Key flows

1. **Startup**: `index.ts` exposes Blockly and arduinoGenerator on `window`, then loads the app (e.g. `BlocklyApplication`). Block definitions (categories/blockly), block types (block_types_registry), and standard block config (configure_standard_blocks) are applied when Blockly is ready.
2. **Workspace inject**: the demo shell (`demo/blockly_application_shell.ts`) creates a `UcBlocklyWorkspaceManager`, injects the workspace (toolbox, theme, options), restores saved blocks if any, and registers plugins (minimap, backpack, search, etc.). Hosts can also use `createUcBlockly()` / `UcBlocklyWorkspaceManager` from `µcBlockly/core` without the demo shell.
3. **Code generation**: User clicks "Generate code" (or equivalent) → `arduinoGenerator.workspaceToCode(workspace)` walks the block tree and calls the per-block generators in **generators/arduino/blocks/**.
4. **Language**: `language.ts` and **languages/** handle UI language and toolbox translation; URL param and dropdown drive the current language.

1.5. Where to change what

Goal	Main files
Add a new block (UI)	`categories/blockly/<category>.ts`, then register in `categories/blockly/index.ts`
Add block type / compatibility	`generators/arduino/block_types/` + `block_types.ts`, then `block_types_registry.ts` — see TYPES_ET_COMPATIBILITES.md
Generate code for a block	`generators/arduino/blocks/<category>.ts` + register in `generators/arduino.ts`
Change toolbox structure	`toolbox.ts` (basic_toolbox, getToolboxForCurrentBoard)
Add a board	`boards.ts` (BOARD_PROFILES + helpers)
New UI language	`languages/languageMap.ts` + new file `languages/<code>.ts`, merge messages
Theme / colours	`options.ts`, `theme_black_and_white.ts`, `theme_seshat.ts`, `ui_palette_theme.ts`
Accessibility (shell)	`shell_a11y.ts`, `a11y_labels.ts`, ACCESSIBILITY.md

Goal	Main files
Patches to Blockly core	`blockly_patches.ts`

1.6. Build, quality, and docs

Command	Purpose
`npm run dev`	Development server with hot reload (Webpack)
`npm run build`	Production build (TypeScript + Webpack)
`npm run build:lib`	TypeScript compile only → `build/`
`npm run lint`	ESLint on `src/` (auto-fix)
`npm run test`	Generator and Blockly@rduino converter tests (`tests/`)
`npm run check-translations`	Validate language files consistency
`npm run convert-rduino`	Convert Blockly@rduino workspace XML to µcBlockly JSON
`npm run generate-favicon`	Regenerate favicons from `public/assets/logo_µcBlockly.png`
`npm run extract-functions`	Regenerate `docs/functions-list.md` from exported symbols
`npm run docs`	Generate **TypeDoc** API site in `docs/api/` (TYPEDOC.md, typedoc.json)

1.6.1. Static assets (''public/'')

- **`public/assets/logo_µcBlockly.png`** — application logo (also favicon source).
- **`public/index.html`** — shell page; favicon links point to 48×48 PNG and ICO in `public/assets/`.
- Other assets (CSS, fonts, Blockly media) live under `public/` and are copied or referenced by Webpack.

Keeping comments, headers, and this architecture doc up to date will help new contributors navigate and improve the project.