86Duino on PlatformIO — Quick Start

From zero to compiling + uploading 86Duino boards in VS Code, in about 15 minutes.

Download: get the latest 86Duino on PlatformIO package from https://www.qec.tw/software/

What's in the box

After unzipping, you get a `PlatformIO-86Duino-501/` folder containing:

- **Complete 86Duino toolchain** (DJGPP GCC 8.3, ~393 MB)
- **86Duino Arduino core + libraries** (libraries, 86EVA and 86HMI synced with the 86Duino Coding 501 IDE build of 2026-07-24)
- **v86dude uploader** (the official 86Duino flash tool)
- **PlatformIO platform definition + SCons builder** (translates the IDE's `platform.txt` into something PlatformIO understands)
- **Example projects**: `examples/blink/` (86Duino ONE), `examples/blink_qecm070t/` (QEC M070T), `examples/sketch_apr29a/` (LVGL HMI demo)
- This `QUICK_START.md`

All 86Duino-specific components are bundled — the standalone 86Duino IDE is not required. (VS Code and the PlatformIO IDE extension are still prerequisites; the first build also auto-downloads PlatformIO's internal `tool-scons` package.)

System requirements

Item	Requirement
Operating system	Windows 10 / 11 (64-bit)
Disk space	~500 MB (after unzip)
Editor	VS Code (recommended)
Required extension	PlatformIO IDE
Windows setting	**Developer Mode MUST be enabled** (see Step 2 for why)

> **Linux / macOS are not supported** — the 86Duino toolchain and v86dude ship as Windows binaries only.

Step 1 — Unzip to a stable location

Unzip somewhere you won't move later, e.g.:

```
C:\86duino-pio\PlatformIO-86Duino-501\
```

Do NOT unzip onto the Desktop or into Downloads — PlatformIO stores absolute paths internally; moving the folder later means re-running `install.ps1` to repair the links (it works, but it's extra work).

Expected directory layout after unzip:

```
PlatformIO-86Duino-501\
├── platform-86duino\
├── framework-arduino86duino\
```

```
├── toolchain-gcc-djgpp\
├── tool-86duino-uploader\
├── 86EVA\               ← EtherCAT config tool (EcConfig) + QEC firmware images
├── 86HMI\               ← 86HMI editor
├── examples\
│   ├── blink\
│   ├── blink_qecm070t\
│   ├── sketch_apr29a\
│   ├── TaiwanExcellenceDemo-m070t_v1r3\
│   └── etg7200_v1r2\
├── scripts\
│   ├── install.ps1
│   ├── new-project.ps1   ← create a new project in 3 questions (see below)
│   └── setup-windows.ps1
├── QUICK_START.md   ← (this file)
├── README.md
└── RECON_REPORT.md
```

Step 2 — Enable Windows Developer Mode (required!)

When PlatformIO installs the platform and packages, it creates **symbolic links (symlinks)** pointing into the unzipped folder. This lets you edit the builder code and see the changes immediately. Windows requires admin privileges to create symlinks by default, but Developer Mode lets unprivileged users create them too.

How to enable:

1. **Settings → Privacy & security → For developers**
2. Toggle **Developer Mode** ON
3. (You may need to reboot once)

> If Developer Mode is off, `install.ps1` will abort at the live-link verification step and tell you to turn it on.

Step 3 — Install VS Code and the PlatformIO IDE extension

1. Install VS Code.
2. Open VS Code, go to **Extensions** in the left sidebar (Ctrl+Shift+X).
3. Search for **PlatformIO IDE** and click **Install**.
4. Wait for the bottom-right status to show `PlatformIO: Installing PlatformIO Core` and **let it finish** (2–5 minutes depending on network speed).
5. After it's done, **restart VS Code once** (`Ctrl+Shift+P` → `Developer: Reload Window`).

Step 4 — Open the PlatformIO Core CLI terminal

In VS Code:

1. Press `Ctrl+Shift+P` to open the command palette.
2. Type `PlatformIO: Open PlatformIO Core CLI` and press Enter.

A new PowerShell terminal opens at the bottom — **this terminal has the `pio` command on its PATH**. Verify:

```
pio --version
```

You should see something like `PlatformIO Core, version 6.1.x`. If you see `'pio' is not recognized as an internal or external command`, the PlatformIO Core install from Step 3 hasn't finished yet or VS Code hasn't been restarted — go back to Step 3.

Step 5 — Run install.ps1 (one-shot setup)

In the PlatformIO CLI terminal you just opened:

```
cd C:\86duino-pio\PlatformIO-86Duino-501
.\scripts\install.ps1
```

If you get an **execution policy** error (message contains "cannot be loaded because running scripts is disabled on this system"), unblock the current session and retry:

```
Set-ExecutionPolicy -Scope Process Bypass -Force
.\scripts\install.ps1
```

`-Scope Process` only affects the current terminal — **it does NOT change your system-wide policy** — and reverts when you close the terminal.

Expected tail of the output:

```
[OK] Toolchain (DJGPP GCC 8.3)
[OK] Arduino framework + libs
[OK] Uploader (v86dude)
[OK] Platform (86duino)

[*] Verifying live-link is wired up...
    Found PIO link: platform-86duino.pio-link -> C:\86duino-pio\PlatformIO-86Duino-501\platform-86duino
[OK] Live-link verified ...

============================================================
 Installation complete!
============================================================
```

If verification fails with `doesn't contain heartbeat`, Windows didn't grant the symlink permission. Go back to **Step 2** and enable Developer Mode, then re-run.

Step 6 — Test compilation (no board required)

```
cd .\examples\blink_qecm070t
pio run
```

The first run will auto-download the `tool-scons` package (used internally by PIO), which takes a few seconds. When it finishes you should see:

```
RAM:   [          ]   0.2% (used 252180 bytes from 134217728 bytes)
Flash: [=         ]   5.0% (used 211716 bytes from 4194304 bytes)
======================== [SUCCESS] Took 16 seconds ========================
```

The `[SUCCESS]` line means **the toolchain, the framework, and the SCons builder all work end-to-end**.

Step 7 — Connect the board and upload

1. Connect the 86Duino board via USB.
2. Open Windows **Device Manager** → **Ports (COM & LPT)** and note the board's COM number (e.g. `COM6`).
3. Alternatively run `pio device list` in the PlatformIO CLI.
4. Upload (replace `COM6` with your actual port):

```
pio run -t upload --upload-port COM6
```

Expected output:

```
"v86dude" "COM6" 20 "...\program.exe" "QEC7"
Waiting the board booting ********** Done
Bootloader version: Niitaka V1.1.1
Selected Board: QEC M070T
Target Board: QEC M070T
BIOS version: Guava 1.00
Uploading the binary sketch *********########################## Done
======================== [SUCCESS] Took 14 seconds ========================
```

The LED on pin 13 should start **blinking once per second**. **End-to-end done.**

Everyday commands

All commands are run from **inside the project folder** (the one containing `platformio.ini`), in the PlatformIO Core CLI terminal (Step 4).

Build

```
# Build only (no board required) — verifies the code compiles
pio run

# Verbose build — prints the full compiler/linker command lines.
# Use this when reporting a build problem.
pio run -v
```

- `pio run` only builds by default — **it does not auto-upload**. To flash the board you MUST add `-t upload`.
- The first build compiles the Arduino core + libraries and is slow (~15 s); later builds only recompile files you changed.

Upload (flash to the board)

```
# Build (if needed) + flash. Auto-detects a PL2303 COM port when
# upload_port is not set.
pio run -t upload

# Pin the port explicitly (check the number in Device Manager or `pio device list`)
pio run -t upload --upload-port COM6
```

- Upload builds automatically first — it only rebuilds when you've actually changed source, otherwise it reuses the existing `program.exe`.
- Tip: set `upload_port = COM6` in `platformio.ini` once, and plain `pio run -t upload` works from then on. Or run `pio run -t select-port` to pick from a menu.

Serial monitor

```
# Open the Serial Monitor to see Serial.println output
pio device monitor --port COM6 --baud 115200
```

- Press `Ctrl+C` to close the monitor.
- If only one COM port exists, `pio device monitor` alone is enough — the wizard already wrote `monitor_speed = 115200` into `platformio.ini`, so the baud rate is picked up automatically.
- The monitor and the uploader can't hold the same COM port at once — close the monitor before `-t upload`, then reopen it.

Clean

```
# Remove compiled objects for the current environment
pio run -t clean

# Wipe the ENTIRE .pio build folder (objects + caches)
pio run -t fullclean
```

- Reach for `fullclean` after the platform / framework / libraries have been updated — stale objects built against old headers cause `undefined reference` link errors.

Board / port management (86Duino custom targets)

```
pio run -t select-board        # pick a board from a menu, writes board = ... to platformio.ini
pio run -t select-port         # pick a PL2303 COM port, writes upload_port = ...
pio run -t reset-board         # reset the board via resetTool.exe
```

See **Custom targets (IDE feature parity)** below for the full list, including `86eva`, `86hmi` and `upload-bootloader`.

Diagnostics

```
pio device list      # all COM ports, with VID:PID (PL2303 = 067B:xxxx)
pio pkg list -g      # verify the packages installed by install.ps1 (platform + 5 tools)
pio system info      # PIO core / Python versions — include when reporting problems
```

Custom targets (IDE feature parity)

These replicate the 86Duino IDE menu functions. Run them from inside any project directory (they also appear as buttons under **PlatformIO sidebar → Project Tasks → Platform** in VS Code):

```
pio run -t select-board        # list all 86Duino boards, write choice to platformio.ini
pio run -t select-port         # list PL2303 COM ports, write upload_port to platformio.ini
pio run -t reset-board         # reset the board via resetTool.exe
pio run -t upload-bootloader   # flash the full bootloader/firmware image (v86dude cmd 34, QEC/CYF only)
pio run -t 86eva               # launch 86EVA (EcConfig) EtherCAT configurator
pio run -t 86hmi               # launch the 86HMI editor
```

Notes:

- `select-port` / `reset-board` / `upload-bootloader` need the board connected via USB.
- `86eva` and `86hmi` require the corresponding packages, which `install.ps1` installs automatically.

Starting your own project

Use the bundled wizard `new-project.ps1` — one command, three questions (project name, board, open-in-VS-Code):

```
cd C:\path\to\PlatformIO-86Duino-501

# If PowerShell refuses to run the script ("running scripts is disabled on
# this system"), unblock the current terminal first — this affects ONLY the
# current session and reverts when you close it:
Set-ExecutionPolicy -Scope Process Bypass -Force

.\scripts\new-project.ps1
```

The wizard reads the board list live from `platform-86duino\boards\`, then generates a ready-to-build project with the same layout as the bundled examples:

```
my_first_app\
├── platformio.ini            ← platform/board/upload_protocol/monitor_speed pre-filled
├── .gitignore                ← ignores .pio and machine-generated .vscode files
├── .pio\                     ← build output (filled on first `pio run`)
├── .vscode\
│   └── extensions.json       ← recommends the PlatformIO IDE extension
└── src\
    └── main.ino              ← setup()/loop() skeleton with Serial.println
```

> `.vscode\c_cpp_properties.json` and `launch.json` are auto-generated by PlatformIO when you open the folder in VS Code — they are machine-specific, so the wizard doesn't create them and the `.gitignore` excludes them.

Non-interactive form (for SOPs / batch setup):

```
.\scripts\new-project.ps1 -Path C:\work\my_app -Board 86duino_qecm070t -NoCode
```

Then just:

```
cd C:\work\my_app
pio run                                 # compile
pio run -t upload --upload-port COM6    # compile + flash
```

> Prerequisite: `install.ps1` has been run once on this machine (Step 5 above), so that `platform = 86duino` resolves.

Two rules when editing `platformio.ini`

> ⚠ **NEVER use `platform = file://...path...` in `platformio.ini`!** PlatformIO handles `file://` by **copying the entire platform into its cache and never refreshing**. Any changes you make to the builder code afterwards will be invisible to PIO. Always use `platform = 86duino` (which resolves to the global symlink that `install.ps1` set up) — that way edits to the platform code take effect immediately. The wizard already writes it this way — don't change it.
>
> ⚠ **Servo86 / AIServo86 exception** — these two libraries' headers contain code like `bool load(const String &dir, const String &fname = NULL)` which **strict GCC 8.3 rejects** (`NULL` is `int(0)`, and `String` has both `String(const char*)` and `String(const __FlashStringHelper*)` constructors, making the conversion ambiguous). If your sketch doesn't `#include` them → no impact. If it does → you'll get a compile error. The fix is either to patch the library header (change `= NULL` to `= String()`) or use a different library.

Adding new files to your project

Drop the files into the `src\` folder — **no need to edit `platformio.ini`**. PIO automatically scans `src\*.cpp`, `src\*.c`, `src\*.ino`, `src\*.h`.

```
my_first_pio_project\
├── platformio.ini
├── src\
│   ├── my_app.ino           ← main sketch (setup/loop)
│   ├── motor_control.cpp    ← new module
│   ├── motor_control.h
│   └── ...
├── lib\                     ← your own reusable libraries go here (optional)
│   └── MyHelper\
│       ├── MyHelper.cpp
│       └── MyHelper.h
└── data\                    ← 86EVA / 86HMI resources (created by the tools; optional)
    └── _project.hpj
```

Using the 86Duino bundled libraries — zero configuration

All libraries that ship with 86Duino IDE 501 (SCoop, Ethercat, LVGL86, Audio, Modbus, SD, EEPROM, Wire, SPI, Encoder, Time86, TimerOne, CANBus, Rosserial86, IRremote, and many more) are bundled inside the framework. **Just `#include` and they work.** Example:

```
#include <Arduino.h>
#include "Ethercat.h"
#include <SCoop.h>
#include <Wire.h>
#include <Motion86.h>      // auto-linked via LDF + already-force-built TimerWDT
#include <LVGL.h>

EthercatMaster master;
// ...
```

`pio run` will automatically:

1. Find the headers via the eager-expanded CPPPATH
2. Pre-compile the 10 FORCE_BUILD_LIBS to resolve transitive symbols from toolchain `.a` files (e.g. `TimerWatchdogTimer::*`)
3. Compile the larger libraries (LVGL86 / Audio / Motion86 / Ethercat) into `.a` archives via PIO LDF based on your sketch's `#include` chain
4. Auto-scan all `lib_link_para.txt` files at builder startup and append the declared `-l` flags (`-lecat`, `-lmotion`, `-lavcodec`, etc.) to LIBS, pointing at the toolchain's precompiled `.a` files
5. Link everything using `--start-group / --end-group`; GCC's `--gc-sections` strips unused symbols

→ You won't need `lib_deps` 90% of the time.

When DO you need `lib_deps`?

- You want a third-party library from the PlatformIO Registry (e.g. ArduinoJson, PubSubClient) — not an 86Duino-bundled one
- Your sketch doesn't directly `#include` an 86Duino library, but a function you call transitively depends on one, AND that library is not in our FORCE_BUILD_LIBS list → add it to `lib_deps`
- You want to pin a specific version of a library

```
lib_deps =
    bblanchon/ArduinoJson@^6.21
    knolleary/PubSubClient
```

Caveat: **third-party libraries must compile for i586 (32-bit x86 running on DOS)**. Many Arduino libraries assume AVR or ARM and won't compile cleanly. If you hit a compile error, share the log.

Available board definitions (13 total, from 86Duino IDE 501)

Vortex86EX family (Arduino Leonardo-style, 1200 bps upload trick)

Board ID	IDE display name
`86duino_one`	86Duino ONE
`86duino_zero`	86Duino ZERO
`educake`	86Duino EduCake
`hmi043t`	86Duino QEC HMI 043T

Vortex86EX2 family (dual-core, QEC industrial line, UART upload)

Board ID	IDE display name
`86duino_qec`	QEC M043T
`86duino_qecm070t` ✓	QEC M070T (7" touch)
`86duino_qecm090t`	QEC M090T (9" touch)
`86duino_qecm150t`	QEC M150T (15" touch)
`86duino_qecppcm090`	QEC PPCM090
`86duino_qecppcm104`	QEC PPCM104
`86duino_qecppcm150`	QEC PPCM150
`86duino_qecm01`	QEC M01
`86duino_qecm02`	QEC M02

> ✓ marks boards verified end-to-end (build + upload) on real hardware. The other boards share the same JSON structure and should work in theory — please share full `pio run -v` output for any issues on first use.

Implementation architecture (for developers / maintainers)

> Regular users don't need to read this section. It's only relevant if you're modifying the builder, adding board support, or debugging link errors.

Why not just "build all bundled libraries into `.a`"?

That was the initial approach (eager-build-everything), but it hit a wall:

Two 86Duino libraries contain GCC-incompatible syntax (`bool load(const String &dir, const String &fname = NULL)`):

- `libraries/Servo86/Servo86.h`
- `libraries/AIServo86/AIServo86.h`

These compile fine in the 86Duino IDE (older GCC + different flags) but fail on our stock GCC 8.3 with strict flags. The `NULL → const String` conversion is ambiguous because `String` has both `String(const char*)` and `String(const __FlashStringHelper*)` constructors. **A sketch that doesn't use Servo86 / AIServo86 should not be punished for this upstream bug.**

→ Final design: pre-build only a curated short list, defer everything else to LDF.

Four-layer library resolution

The library-resolution logic spans four layers — three in `platform-86duino\builder\frameworks\arduino.py`, plus the `lib_link_para.txt` scan in the main platform script:

```
┌──────────────────────────────────────────────────────────┐
│ Layer 1 — CPPPATH eager expansion                        │
│                                                          │
│ for entry in os.listdir(LIBS_DIR):                       │
│     extra_cppath += [entry, entry/src, entry/utility]    │
│ env.Append(CPPPATH=extra_cppath)                         │
│                                                          │
│ → Any #include resolves at the preprocessor level.       │
│ → Path-only — does NOT compile source. Servo86.h is in   │
│   CPPPATH but won't be built unless a sketch #includes it.  │
└──────────────────────────────────────────────────────────┘
                            │
┌──────────────────────────────────────────────────────────┐
│ Layer 2 — FORCE_BUILD_LIBS curated list (10 libs)        │
│                                                          │
│ FORCE_BUILD_LIBS = [                                     │
│     "TimerWDT", "TimerOne", "MsTimer2", "Time86",        │
│     "SoftwareSerial", "SPI", "Wire", "EEPROM",           │
│     "SCoop", "Encoder",                                  │
│ ]                                                        │
│ for lib in FORCE_BUILD_LIBS:                             │
│     archive = env.BuildLibrary(...)                      │
│ env.Prepend(LIBS=fw_archives)                            │
│                                                          │
│ → Resolves "toolchain `.a` uses it transitively, LDF can't  │
│   see it" — e.g. libmotion.a → TimerWatchdogTimer::*     │
│ → Logs "Force-built 10 transitive-dep libraries" to stderr  │
└──────────────────────────────────────────────────────────┘
                            │
┌──────────────────────────────────────────────────────────┐
│ Layer 3 — PIO LDF (built-in)                             │
│                                                          │
│ → Scans the sketch's #include chain                      │
│ → Looks up libs under framework-arduino86duino/libraries/   │
│ → Compiles to .a and auto-links                          │
│                                                          │
│ → Handles big libs like LVGL86, Audio, Motion86, Ethercat,  │
│   SD. Servo86 / AIServo86 are also handled here — and only  │
│   trigger their compile failure if the sketch #includes  │
│   them. This is the intended behavior.                   │
└──────────────────────────────────────────────────────────┘
                            │
┌──────────────────────────────────────────────────────────┐
│ Layer 4 — lib_link_para.txt auto-scan (in platform main.py) │
│                                                          │
│ for each libraries/*/lib_link_para.txt:                  │
│     parse `-l` flags                                     │
```

```
│     env.Append(LIBS=[...])                                  │
│                                                             │
│ → 86Duino convention: a library declares "I need to link    │
│   xxx.a" in this file. The builder reads them all.          │
│ → e.g. Audio/lib_link_para.txt contains `-lavcodec          │
│   -lswresample`. All flags go into LIBS; GCC --gc-sections  │
│   strips what isn't actually referenced.                    │
└─────────────────────────────────────────────────────────────┘
```

Why four layers?

Layer	Problem it solves
L1 CPPPATH eager expansion	86Duino libraries cross-`#include` each other without relative paths, e.g. Ethercat.h does `#include "scheduler.h"`. PIO LDF's default chain mode can't see these cross-library includes.
L2 FORCE_BUILD_LIBS	The toolchain ships precompiled `libmotion.a` and similar that reference 86Duino-side classes. LDF can't see references inside `.a` archives, so these libs MUST be pre-built and put in LIBS.
L3 PIO LDF	Big libraries (LVGL = dozens of `.c` files, Audio = full ffmpeg) shouldn't be compiled every time. Only build when actually used.
L4 lib_link_para.txt	Pre-existing 86Duino convention. Libraries declare "I need link flag `-lxxx`" and the builder picks them up automatically.

Maintenance — adding a new transitive-dep lib to FORCE_BUILD_LIBS

If a new sketch hits `undefined reference to 'SomeClass::method'` where `SomeClass` is used by a toolchain precompiled `.a` (not directly by the sketch), the fix is:

```
# platform-86duino\builder\frameworks\arduino.py — around line 105
FORCE_BUILD_LIBS = [
    ...,
    "NewLibName",   # ← add here
]
```

Save → `pio run` picks it up immediately (symlink install means edits are live).

Maintenance — adding a new board

1. Drop a new JSON in `platform-86duino\boards\`. Copy `86duino_qecm070t.json` as a template and edit:
2. `name` / `description`
3. `build.variant` (must correspond to `framework-arduino86duino\variants\<variant>\`)
4. `build.boardname` (must match the first column in `tool-86duino-uploader\interface.v86`)
5. `build.linkerex2` (EX2 boards: `-lipc -lproc -lvdpm`; EX boards: leave empty)
6. `upload.protocol` (EX boards: `avr109`; EX2 boards: `uart`)
7. `vendor.usb` (EX boards: `0525:b4a7`; EX2 boards: `067B:23A3`)
8. Add a row to the "Available board definitions" table in `QUICK_START.md`.
9. Validate on real hardware.

Maintenance — what does install.ps1 actually do?

Key steps:

1. Locate `pio.exe` (PATH → `%USERPROFILE%\.platformio\penv` → Python Scripts global glob).
2. **Purge any stale installs** — the `file://` cache copy is particularly toxic: once installed, PIO never refreshes it.
3. Run `pkg install -g symlink://<absolute path>` for all four packages (toolchain, framework, uploader, platform).
4. **Verify the live-link** — parse `~/.platformio/platforms/*86duino*.pio-link` (which may be a plain path OR JSON of the form `{"spec":{"uri":"symlink://..."}}`) and use `Select-String` to find the `[86duino] platform main.py loaded` heartbeat. This confirms PIO is actually using a symlink, not a Windows copy.
5. If verification fails → abort with a message telling the user to enable Windows Developer Mode.

Troubleshooting

`pio --version` reports `'pio' is not recognized as an internal or external command`

→ You're not in the PlatformIO Core CLI terminal. Go back to Step 4: `Ctrl+Shift+P` → `PlatformIO: Open PlatformIO Core CLI`.

`install.ps1` reports "execution policy disabled"

→ Run `Set-ExecutionPolicy -Scope Process Bypass -Force` and retry.

`install.ps1` aborts at verification with `doesn't contain heartbeat`

→ Windows didn't allow symlink creation and PIO installed the platform as a copy instead. Go to **Settings → Privacy & security → For developers → Developer Mode** and toggle it ON, then re-run `install.ps1`.

Compile error: `UnknownPackageError: Could not find the package with '86duino' requirements`

→ Global package not installed. Run `.\scripts\install.ps1` from the `PlatformIO-86Duino-501` root.

Compile error: fatal error `xxx.h: No such file or directory`

In theory this shouldn't happen — the framework already adds every library's include path to CPPPATH. If you see it, one of:

1. The header is not bundled with the framework (third-party or custom). Drop it into the sketch's `src/` or `lib/` folder.
2. The framework installation is corrupted (rare). Re-run `.\scripts\install.ps1`.

Link error: `undefined reference to 'some 86Duino class'`

Example: `undefined reference to 'SomeLib::method(...)'`. Two possibilities:

(A) The library should have been picked up by LDF but your sketch doesn't `#include` it → LDF missed it. Fix: add `lib_deps = <LibName>` in `platformio.ini`, or add the lib name to `FORCE_BUILD_LIBS` in `platform-86duino\builder\frameworks\arduino.py`.

(B) The reference comes from a toolchain `.a` file used transitively (e.g. `libmotion.a` calls `TimerWatchdogTimer`) → This is exactly what FORCE_BUILD_LIBS is for. If you've discovered a new transitive dependency, add it to the list:

```
# platform-86duino\builder\frameworks\arduino.py
FORCE_BUILD_LIBS = [
    "TimerWDT", "TimerOne", "MsTimer2", "Time86",
    "SoftwareSerial", "SPI", "Wire", "EEPROM",
    "SCoop", "Encoder",
    "YourNewLib",   # ← add here
]
```

Save and re-run `pio run` (symlink install — changes are live immediately).

Servo86 / AIServo86: `conversion from 'int' to 'const String' is ambiguous`

This is an **upstream GCC-compatibility bug in the library** — the header writes `bool load(const String &dir, const String &fname = NULL)`, which strict GCC 8.3 rejects (`NULL` is `int(0)`, and both `String(const char*)` and `String(const __FlashStringHelper*)` are candidates for the conversion).

We intentionally **do not** include Servo86 / AIServo86 in FORCE_BUILD_LIBS, so as long as your sketch doesn't `#include "Servo86.h"` or `"AIServo86.h"`, you won't see this error.

If you really need Servo86 / AIServo86, two options:

1. **Patch the library header** (permanent fix): open `framework-arduino86duino\libraries\Servo86\Servo86.h` and change every `= NULL` to `= String()` (affected lines: 334, 378 × 2, 400). Same for AIServo86. Edits take effect immediately because the framework is also a symlink install.
2. **Use the standard Arduino Servo library**: `#include <Servo.h>` (also bundled in `libraries\Servo\` and free of this bug).

Edited `platform-86duino\builder\main.py` but `pio run` ignores my changes

→ Your `platformio.ini` is probably still using `platform = file://...`. PIO handles `file://` by **copying the platform into cache on install and never refreshing it**. Switch to `platform = 86duino` (which requires `install.ps1` to have run, setting up the global symlink), then:

```
pio run -t fullclean
pio run
```

Confirm PIO is using the live version:

```
$plat = Get-ChildItem "$env:USERPROFILE\.platformio\platforms\" -Force | Where-Object { $_.Name -like
"*86duino*" }
$plat | Select-Object Name, FullName, Length
```

You should see a `platform-86duino.pio-link` entry. Open it (it's JSON) — `spec.uri` should point to your unzipped `PlatformIO-86Duino-501` folder.

Compile error: `'cmd_t' has not been declared` (some line and column)

→ PIO's `.ino` preprocessor inserted a forward declaration above the `#include` that introduces the type. Two fixes:

1. **Quick**: move the header that defines the type to the very top of the sketch, before any other `#include`.
2. **Proper**: rename the sketch from `.ino` to `.cpp` and add `#include <Arduino.h>` at the top — this bypasses PIO's `.ino` preprocessing entirely.

Upload error: `No found the interface on selected board`

→ Either your `board =` value in `platformio.ini` is wrong, OR the board JSON is missing the `build.boardname` field. Open `platform-86duino\boards\<your_board>.json` and verify there's a `"boardname"` field matching the first column of `tool-86duino-uploader\interface.v86` (e.g. `QEC7`, `ONE`, `ZERO`, `EDUCAKE`).

Upload error: `... Error command` or `Bootloader: Receive a unavailable command`

→ This usually means the `board =` value in `platformio.ini` doesn't match the connected hardware — run `pio run -t select-board` and pick the correct model, then retry. For bootloader/firmware updates, don't set command codes manually; use the dedicated target, which handles the correct v86dude command internally:

```
pio run -t upload-bootloader
```

Upload error: `Open src file: fail, fp = NULL` / `the uploaded file is 0 byte`

→ The path to `program.exe` didn't resolve. Make sure `pio run` succeeded first (`.pio/build/<board>/program.exe` must exist).

Can't find the COM port

→ Verify:

1. The USB cable is properly connected (board's power LED is on).
2. Windows **Device Manager** → **Ports** lists the device.

3. If you see an "Unknown device" or a yellow warning triangle, install the Windows PL2303 driver — download it from the QEC Software page: https://www.qec.tw/software/ (under "Resources").

Upload hangs with no progress

→ The board may be stuck in BIOS. Unplug USB, press the reset button, plug back in, and re-upload.

VS Code shows `C/C++ IntelliSense service does not support .INO files`

→ This is a **VS Code editor warning — it does NOT affect PlatformIO compilation or upload**. It only means the editor's IntelliSense (red squiggles, autocomplete) doesn't work on `.ino` files.

Two ways to handle:

1. **Ignore**: dismiss the warning.
2. **Convert sketch from `.ino` to `.cpp`** (full IntelliSense support):
   a. Rename `src\<sketch>.ino` to `src\<sketch>.cpp`.
   b. Add `#include <Arduino.h>` at the top of the file.
   c. If any function outside `setup()` / `loop()` is called from elsewhere, add forward declarations at the top.

Note: after converting to `.cpp` the sketch won't open in the 86Duino IDE. If you need both IDEs to work, stick with `.ino` and ignore the warning.

Command cheatsheet

> **Reminder: `pio run` only builds, never auto-uploads. To flash the board you MUST use `-t upload`.**

```
# Build only (no upload)
pio run

# Build + flash (board must be connected)
pio run -t upload --upload-port COM6

# Verbose output (for debugging)
pio run -v

# Open Serial Monitor
pio device monitor --port COM6 --baud 115200

# List COM ports
pio device list

# Clean build cache (when you hit weird compile errors)
pio run -t clean         # current env only
pio run -t fullclean     # full clean (including LDF cache)

# List installed packages
pio pkg list -g

# Reinstall the platform / packages (when something is broken)
.\scripts\install.ps1
```

Advanced — modifying the builder yourself

The full build logic lives in `platform-86duino\builder\main.py` (a SCons translation of the IDE's `platform.txt`). If you find a compile flag or link library that's wrong, edit this file directly — **because the install is a symlink, your changes take effect immediately, no reinstall needed.**

Verify the live-link is still in place:

```
$plat = Get-ChildItem "$env:USERPROFILE\.platformio\platforms\" -Filter "*86duino*.pio-link"
Get-Content $plat.FullName
```

You should see JSON — `spec.uri` should point to your unzipped `PlatformIO-86Duino-501` folder.

Hit a problem?

Send the maintainer the following three pieces of info:

1. `pio --version` output
2. `pio pkg list -g` output
3. Full `pio run -v` log — capture with `pio run -v 2>&1 | Tee-Object -FilePath build.log`

Need more

Feedback is welcome at info@qec.tw or via the contact form.