86Duino on PlatformIO – クイックスタート

何もない状態から、VS Code で 86Duino ボードのコンパイルとアップロードができるようになるまでを、約 15 分で解説します。

ダウンロード: 最新の 86Duino on PlatformIO パッケージは https://www.qec.tw/software/ から入手してください。

同梱内容

解凍すると、次の内容を含む `PlatformIO-86Duino-501/` フォルダが作成されます:

- **86Duino ツールチェーン一式**(DJGPP GCC 8.3、約 393MB)
- **86Duino Arduino コア＋ライブラリ**(86Duino Coding 501 IDE の 2026-07-24 ビルドと同期したライブラリ、86EVA、86HMI)
- **v86dude アップローダー**(86Duino 公式の書き込みツール)
- **PlatformIO プラットフォーム定義＋SCons ビルダー**(IDE の `platform.txt` を PlatformIO が理解できる形式に変換します)

サンプルプロジェクト:

- `examples/blink/`(86Duino ONE 用)
- `examples/blink_qecm070t/`(QEC M070T 用)
- `examples/sketch_apr29a/`(LVGL HMI デモ)
- **本ドキュメント**(`QUICK_START.md`)

86Duino 専用のコンポーネントはすべて同梱されているため、86Duino IDE 単体は不要です。(ただし VS Code と PlatformIO IDE 拡張機能は引き続き必要です。また初回ビルド時には PlatformIO 内部の `tool-scons` パッケージが自動でダウンロードされます。)

動作環境

項目	要件
OS	Windows 10 / 11(64-bit)
ディスク容量	約 500MB(解凍後)
エディタ	VS Code(推奨)
必須拡張機能	PlatformIO IDE
Windows 設定	**開発者モードを必ず有効にしてください**(理由は Step 2 を参照)

> **Linux / macOS には対応していません** – 86Duino ツールチェーンおよび v86dude は Windows 用バイナリのみが提供されています。

Step 1 — 定まった場所に解凍する

後で移動しない場所に解凍してください。例:

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

デスクトップやダウンロードフォルダには解凍しないでください — PlatformIO は内部で絶対パスを保持するため、フォルダを後で移動すると、リンクを修復するために `install.ps1` を再実行する必要があります(動作はしますが、余計な手間がかかります)。

解凍後に想定されるディレクトリ構成:

```
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 — Windows 開発者モードを有効にする(必須!)

PlatformIO がプラットフォームとパッケージをインストールする際、解凍したフォルダを指す**シンボリックリンク(symlink)**が作成されます。これにより、ビルダーのコードを編集すると変更内容が即座に反映されるようになります。Windows では既定でシンボリックリンクの作成に管理者権限が必要ですが、開発者モードを有効にすると管理者権限のないユーザーでも作成できるようになります。

有効化の手順:

1. **設定** → **プライバシーとセキュリティ** → **開発者向け**
2. **開発者モード**を ON に切り替える
3. (再起動が必要になる場合があります)

> 開発者モードが無効の場合、`install.ps1` はライブリンクの検証ステップで失敗し、有効化するよう案内が表示されます。

Step 3 — VS Code と PlatformIO IDE 拡張機能をインストールする

1. **VS Code** をインストールします。
2. VS Code を開き、左サイドバーの**拡張機能**(Ctrl+Shift+X)を開きます。
3. **PlatformIO IDE** を検索し、**Install** をクリックします。
4. 右下のステータスに **PlatformIO: Installing PlatformIO Core** と表示され、**完了するまで待ちます**(ネットワーク速度により 2〜5 分)。
5. 完了したら、**一度だけ VS Code を再起動します**(`Ctrl+Shift+P` → `Developer: Reload Window`)。

Step 4 — PlatformIO Core CLI ターミナルを開く

VS Code 上で:

1. `Ctrl+Shift+P` を押してコマンドパレットを開きます。
2. `PlatformIO: Open PlatformIO Core CLI` と入力して Enter を押します。

画面下部に新しい PowerShell ターミナルが開きます — **このターミナルには `pio` コマンドの PATH が通っています。**確認:

```
pio --version
```

`PlatformIO Core, version 6.1.x` のような表示が出れば成功です。もし `'pio' is not recognized as an internal or external command` と表示される場合は、Step 3 の PlatformIO Core のインストールがまだ完了していないか、VS Code が再起動されていません — Step 3 に戻ってください。

Step 5 — install.ps1 を実行する(初回セットアップ)

先ほど開いた PlatformIO CLI ターミナルで:

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

実行ポリシーエラー("cannot be loaded because running scripts is disabled on this system" というメッセージ)が出た場合は、現在のセッションのみブロックを解除して再実行してください:

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

`-Scope Process` は現在のターミナルにのみ影響します — **システム全体のポリシーは変更されず**、ターミナルを閉じると元に戻ります。

出力の末尾の例:

```
[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!
============================================================
```

`doesn't contain heartbeat` という表示とともに検証に失敗した場合は、Windows がシンボリックリンクの作成権限を許可していません。Step 2 に戻って開発者モードを有効にしてから再実行してください。

Step 6 – テストコンパイル（ボード不要）

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

初回実行時は PIO が内部で使用する `tool-scons` パッケージが自動ダウンロードされるため、数秒かかります。完了すると、次のような表示が確認できます:

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

======================== [SUCCESS] Took 16 seconds ========================
```

`[SUCCESS]` の行が表示されれば、ツールチェーン・フレームワーク・SCons ビルダーがすべて正常に連携して動作していることを意味します。

Step 7 – ボードを接続してアップロードする

1. 86Duino ボードを USB で接続します。
2. Windows の **デバイスマネージャー → ポート（COM と LPT）** を開き、ボードの COM 番号を確認します（例: `COM6`）。
3. あるいは、PlatformIO CLI で次を実行しても確認できます:

```
pio device list
```

4. アップロードします（`COM6` の部分は実際のポート番号に置き換えてください）:

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

出力例:

```
"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 ========================
```

13 番ピンの LED が 1 秒間隔で点滅し始めれば、エンドツーエンドでの動作確認は完了です。

日常的に使うコマンド

すべてのコマンドは、`platformio.ini` を含む**プロジェクトフォルダの中**で、PlatformIO Core CLI ターミナル（Step 4）から実行します。

ビルド

```
# 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` は既定では**ビルドのみ**を行い、自動アップロードはしません。ボードに書き込むには `-t upload` を必ず追加してください。
- 初回ビルドは Arduino コア＋ライブラリのコンパイルを含むため遅く（約 15 秒）、以降のビルドは変更されたファイルのみ再コンパイルされます。

アップロード（ボードへの書き込み）

```
# 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
```

- アップロードは自動的にまずビルドを行いますが、実際にソースを変更した場合のみ再ビルドされ、それ以外は既存の `program.exe` がそのまま使われます。
- Tip: `platformio.ini` に `upload_port = COM6` を一度設定しておけば、以降はそのまま `pio run -t upload` を実行するだけで済みます。または `pio run -t select-port` を実行するとメニューから選択できます。

シリアルモニタ

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

- `Ctrl+C` を押すとモニタを閉じられます。

- COM ポートが 1 つしか存在しない場合は `pio device monitor` だけで十分です – ウィザードが `platformio.ini` に `monitor_speed = 115200` を書き込み済みのため、ボーレートは自動的に反映されます。
- モニタとアップローダーは同じ COM ポートを同時に使用できません – `-t upload` の前にモニタを閉じ、完了後に再度開いてください。

クリーン

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

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

- プラットフォーム／フレームワーク／ライブラリが更新された後は `fullclean` を使ってください – 古いオブジェクトが残っていると `undefined reference` リンクエラーの原因になります。

ボード／ポート管理（86Duino カスタムターゲット）

```
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
```

`upload-bootloader`、`86eva`、`86hmi` を含む完全な一覧は、下記の「カスタムターゲット（IDE 機能パリティ）」を参照してください。

診断

```
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
pio system info        # tools) / PIO core, Python versions — include when reporting
```

カスタムターゲット（IDE 機能パリティ）

これらは 86Duino IDE のメニュー機能を再現したものです。どのプロジェクトディレクトリからでも実行できます（VS Code では **PlatformIO サイドバー** → **Project Tasks** → **Platform** の下にボタンとしても表示されます）:

```
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
```

備考:

- `select-port` / `reset-board` / `upload-bootloader` はボードが USB で接続されている必要があります。
- `86eva` と `86hmi` は対応するパッケージが必要ですが、これは `install.ps1` が自動的にインストールします。

自分のプロジェクトを開始する

同梱のウィザード `new-project.ps1` を使えば、1 つのコマンドと 3 つの質問(プロジェクト名、ボード、VS Code で開くかどうか)だけで新規プロジェクトを作成できます:

```
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
```

ウィザードは `platform-86duino\boards\` からボード一覧をリアルタイムに読み込み、同梱サンプルと同じ構成のビルド可能なプロジェクトを生成します。

```
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` と `launch.json` は、VS Code でフォルダを開いた際に PlatformIO が自動生成します — 環境依存の内容のため、ウィザードでは作成されず `.gitignore` で除外されています。

非対話形式(SOP／バッチセットアップ向け):

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

あとは次を実行するだけです:

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

前提条件: このマシンで一度 `install.ps1` が実行済みであること(前述の Step 5)。これにより `platform = 86duino` が正しく解決されます。

`platformio.ini` を編集する際の 2 つのルール

⚠ **`platformio.ini` で `platform = file://...path...` を絶対に使用しないでください！** PlatformIO は `file://` を、プラットフォーム全体をキャッシュへコピーし、以降は一切更新しない方式で処理します。ビルダーのコードを後から変更しても、その変更は PIO には反映されません。必ず `platform = 86duino`(`install.ps1` がセットアップするグローバルなシンボリックリンクを指します)を使用してください – こうすることでプラットフォームコードへの変更が即座に反映されます。ウィザードも最初からこの形式で書き込みます – 変更しないでください。

⚠ **Servo86 / AIServo86 の例外** – これら 2 つのライブラリのヘッダーには `bool load(const String &dir, const String &fname = NULL)` のようなコードが含まれており、厳格な GCC 8.3 はこれを拒否します(`NULL` は `int(0)` であり、`String` は `String(const char*)` と `String(const __FlashStringHelper*)` の両方のコンストラクタを持つため、変換があいまいになります)。スケッチがこれらを `#include` していなければ影響はありません。`#include` している場合はコンパイルエラーになります。対処法は、ライブラリヘッダーを修正する(`= NULL` を `= String()` に変更する)か、別のライブラリを使用することです。

プロジェクトへの新規ファイル追加

ファイルは `src\` フォルダに配置するだけで済み、**`platformio.ini` を編集する必要はありません。**PIO は `src\*.cpp`、`src\*.c`、`src\*.ino`、`src\*.h` を自動的にスキャンします。

```
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
```

86Duino 同梱ライブラリの利用 – 設定不要

86Duino IDE 501 に同梱されているすべてのライブラリ(SCoop、Ethercat、LVGL86、Audio、Modbus、SD、EEPROM、Wire、SPI、Encoder、Time86、TimerOne、CANBus、Rosserial86、IRremote、その他多数)は、フレームワーク内にあらかじめ組み込まれています。**`#include` するだけで動作します。**例:

```
#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` は次を自動的に行います:

- 先行展開された CPPPATH 経由でヘッダを検出
- トランジティブなシンボルを解決するため、10 個の FORCE_BUILD_LIBS(トールチェーンから提供される `.a`)を事前コンパイル
- サケッチの `#include` チェーンに基づき、より大きなライブラリ(LVGL86 / Audio / Motion86 / Ethercat)を PIO LDF 経由で `.a` アーカイブへコンパイル
- ビルダー起動時に全ての `lib_link_para.txt` ファイルを自動スキャンし、宣言されたフラグ(`-lecat`、`-lmotion`、`-lavcodec` 等)を toolchain の事前コンパイル済み `.a` ファイルを指す形で LIBS に追加
- `--start-group` / `--end-group` を使ってすべてをリンク — GCC の `--gc-sections` が未使用シンボルを削除

→ **90% のケースでは `lib_deps` は不要です。**

`lib_deps` が必要になるのはどんなときですか?

- PlatformIO Registry のサードパーティ製ライブラリ(例: ArduinoJson、PubSubClient)を使いたい場合 — 86Duino 同梱ライブラリではないもの
- スケッチが直接 `#include` していない 86Duino ライブラリに、間接的に依存する関数を呼び出している場合、かつそのライブラリが FORCE_BUILD_LIBS に含まれない場合 → `lib_deps` に追加してください
- ライブラリの特定バージョンを固定したい場合

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

注意: **サードパーティ製ライブラリは i586(DOS 上で動作する 32-bit x86)向けにコンパイルできる必要があります。** 多くの Arduino ライブラリは AVR や ARM を前提としており、そのままではコンパイルできません。コンパイルエラーが発生した場合はログを共有してください。

利用可能なボード定義(86Duino IDE 501 提供、全 13 種)

Vortex86EX ファミリー(Arduino Leonardo 互換、1200bps アップロードトリック)

Board ID	IDE 表示名
86duino_one	86Duino ONE

`86duino_zero`	86Duino ZERO
`educake`	86Duino EduCake
`hmi043t`	86Duino QEC HMI 043T

Vortex86EX2 ファミリー（デュアルコア、QEC 産業用ライン、UART アップロード）

Board ID	IDE 表示名
`86duino_qec`	QEC M043T
`86duino_qecm070t` ✓	QEC M070T（7 インチタッチ）
`86duino_qecm090t`	QEC M090T（9 インチタッチ）
`86duino_qecm150t`	QEC M150T（15 インチタッチ）
`86duino_qecppcm090`	QEC PPCM090
`86duino_qecppcm104`	QEC PPCM104
`86duino_qecppcm150`	QEC PPCM150
`86duino_qecm01`	QEC M01
`86duino_qecm02`	QEC M02

> ✓ は実機でビルド＋アップロードの動作確認が取れているボードを示します。それ以外のボードも同じ JSON 構造を共有しており理論上は動作するはずですが、初めてお使いになった際は問題の有無にかかわらず `pio run -v` の出力全体を共有してください。

実装アーキテクチャ（開発者／保守担当者向け）

> 一般のユーザーはこの章を読む必要はありません。ビルダーの改造、ボードサポートの追加、リンクエラーの調査を行う場合にのみ関係します。

なぜ単純に「同梱ライブラリをすべて `.a` にビルドする」方式にしないのか？

それが当初のアプローチ（すべてを事前にビルドする方式）でしたが、次の壁にぶつかりました:

86Duino のライブラリのうち 2 つに GCC 非互換の構文が含まれています（`bool load(const String &dir, const String &fname = NULL)`）:

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

これらは 86Duino IDE（旧世代の GCC ＋異なるフラグ）ではコンパイルできますが、厳格なフラグを使う標準の GCC 8.3 ではビルドに失敗します。`NULL → const String` の変換は、`String` が `String(const char*)` と `String(const __FlashStringHelper*)` の両方のコンストラクタを持つためあいまいになります。Servo86 / AIServo86 を使わないスケッチが、この上流のバグによって影響を受けるべきではありません。

→ 最終的な設計: 厳選した短いリストのみを事前ビルドし、それ以外は LDF に委ねる。

4 層のライブラリ解決の仕組み

ライブラリ解決のロジックは 4 つの層にまたがっており、そのうち 3 層は `platform-86duino\builder\frameworks\arduino.py` に、残る 1 層はメインのプラットフォームスクリプトでの `lib_link_para.txt` スキャンにあります:

```
┌──────────────────────────────────────────────────────────────┐
│ 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's
|   --gc-sections strips what isn't actually referenced.
└──────────────────────────────────────────────────────────┘
```

なぜ 4 層に分かれているのか？

層	解決する問題
L1 CPPPATH の先行展開	86Duino のライブラリは相対パスを使わずに互いを #include し合っている（例: Ethercat.h は #include "scheduler.h" する）。PIO LDF の既定のチェーンモードでは、このようなライブラリ間の #include を検出できない。
L2 FORCE_BUILD_LIBS	toolchain には 86Duino 側のクラスを参照する libmotion.a 等の事前コンパイル済みライブラリが同梱されている。LDF は .a アーカイブ内部の参照を検出できないため、これらは事前にビルドして LIBS に入れる必要がある。
L3 PIO LDF	大きなライブラリ（LVGL は .c ファイル数十個、Audio は ffmpeg 全体）を毎回コンパイルすべきではない。実際に使われるときのみビルドする。
L4 lib_link_para.txt	既存の 86Duino の慣習。ライブラリが「リンクフラグ -lxxx が必要」と宣言し、ビルダーが自動的にそれを検出する。

保守: FORCE_BUILD_LIBS への新規トランジティブ依存ライブラリの追加

新しいスケッチで `undefined reference to 'SomeClass::method'`（`SomeClass` はスケッチから直接ではなく、toolchain の事前コンパイル済み `.a` から使用されている）が発生した場合、対処方法は次のとおりです:

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

保存すると `pio run` が即座に反映します（シンボリックリンクでのインストールのため、変更はすぐに有効になります）。

保守: 新しいボードの追加

1. `platform-86duino\boards\` に新しい JSON を配置します。`86duino_qecm070t.json` をテンプレートとしてコピーし編集してください:
2. `name` / `description`

3. `build.variant`（`framework-arduino86duino\variants\<variant>\` と対応している必要があります）
4. `build.boardname`（`tool-86duino-uploader\interface.v86` の 1 列目と一致している必要があります）
5. `build.linkerex2`（EX2 ボード: `-lipc -lproc -lvdpm`；EX ボード: 空のまま）
6. `upload.protocol`（EX ボード: `avr109`；EX2 ボード: `uart`）
7. `vendor.usb`（EX ボード: `0525:b4a7`；EX2 ボード: `067B:23A3`）
8. `QUICK_START.md` の「利用可能なボード定義」の表に行を追加します。
9. 実機で動作確認を行います。

保守: install.ps1 は実際に何を行っているか

主なステップ:

1. `pio.exe` を特定します（PATH → `%USERPROFILE%\.platformio\penv` → Python の Scripts をグローバルに検索）。
2. 既存の不正なインストールを削除します – `file://` によるキャッシュコピーは特に問題があり、一度インストールされると PIO は二度と更新しません。
3. 4 つのパッケージ（toolchain、framework、uploader、platform）すべてに対して `pkg install -g symlink://<absolute path>` を実行します。
4. **ライブリンクを検証します** – `~/.platformio/platforms/*86duino*.pio-link`（プレーンなパス、または `{"spec":{"uri":"symlink://..."}}` 形式の JSON のいずれか）を解析し、`Select-String` を使って `[86duino] platform main.py loaded` というハートビートを検索します。これにより PIO が実際にシンボリックリンクを使用しており、Windows のコピーではないことを確認します。
5. 検証に失敗した場合 → Windows の開発者モードを有効にするようメッセージを表示して中断します。

トラブルシューティング

`pio --version` が `'pio' is not recognized as an internal or external command` と表示される

→ PlatformIO Core CLI ターミナルにいません。Step 4 に戻ってください: `Ctrl+Shift+P` → `PlatformIO: Open PlatformIO Core CLI`

`install.ps1` が "execution policy disabled" と表示する

→ `Set-ExecutionPolicy -Scope Process Bypass -Force` を実行してから再試行してください。

`install.ps1` が `doesn't contain heartbeat` で検証に失敗する

→ Windows がシンボリックリンクの作成を許可せず、PIO がプラットフォームをコピーとしてインストールしてしまいました。**設定** → **プライバシーとセキュリティ** → **開発者向け** → **開発者モード**に移動して ON に切り替え、`install.ps1` を再実行してください。

コンパイルエラー: `UnknownPackageError: Could not find the package with '86duino' requirements`

→ グローバルパッケージがインストールされていません。`PlatformIO-86Duino-501` のルートから `.\scripts\install.ps1` を実行してください。

コンパイルエラー: `fatal error xxx.h: No such file or directory`

理論上これは発生しないはずです – フレームワークがすべてのライブラリのインクルードパスをあらかじめ CPPPATH に追加しているためです。もし発生した場合、次のいずれかが考えられます:

1. そのヘッダがフレームワークに同梱されていない(サードパーティ製またはカスタム)。スケッチの `src/` または `lib/` フォルダに追加してください。
2. フレームワークのインストールが破損している(まれ)。`.\scripts\install.ps1` を再実行してください。

リンクエラー: `undefined reference to 'some 86Duino class'`

例: `undefined reference to 'SomeLib::method(...)'`。2 つの可能性があります:

(A) ライブラリが LDF によって検出されるはずだが、スケッチが `#include` していない

`platformio.ini` に `lib_deps = <LibName>` を追加するか、`platform-86duino\builder\frameworks\arduino.py` の FORCE_BUILD_LIBS にライブラリ名を追加してください → LDF が見逃していたのが原因です。

(B) 参照元が toolchain の `.a` ファイルで、間接的に使用されている

(例: `libmotion.a` が `TimerWatchdogTimer` を呼び出している)→ これはまさに FORCE_BUILD_LIBS が対象とするケースです。新しいトランジティブ依存関係を発見した場合は、リストに追加してください:

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

保存して `pio run` を再実行してください(シンボリックリンクでのインストールのため、変更は即座に反映されます)。

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

これはライブラリ内の上流の GCC 互換性バグです – ヘッダが `String &fname = NULL)` と記述しており、厳格な GCC 8.3 はこれを拒否します(`NULL` は `int(0)` であり、`String(const char*)` と `String(const __FlashStringHelper*)` の両方が変換候補になるためです)。

私たちは意図的に Servo86 / AIServo86 を FORCE_BUILD_LIBS に含めていません。そのため、スケッチが `#include "Servo86.h"` または `"AIServo86.h"` をしない限り、このエラーは発生しません。

どうしても **Servo86 / AIServo86** が必要な場合、**2** つの選択肢があります:

1. **ライブラリヘッダーを修正する(恒久的な対策)**: `framework-arduino86duino\libraries\Servo86\Servo86.h` を開き、該当箇所すべての `= NULL` を `= String()` に変更します(該当行: 334、378×2、400)。AIServo86 も同様です。フレームワークもシンボリックリンクでインストールされているため、編集内容は即座に反映され、再インストールは不要です。
2. **標準の Arduino Servo ライブラリを使う**: `#include <Servo.h>`(`libraries\Servo\` にも同梱されており、このバグの影響を受けません)。

`platform-86duino\builder\main.py` を編集したが、`pio run` に変更が反映されない

→ おそらく `platformio.ini` がまだ `platform = file://...path...` を使用しています。PIO は `file://` を、インストール時にプラットフォーム全体をキャッシュへコピーし、以降は更新しない方式で処理します。`platform = 86duino`(`install.ps1` のセットアップによりグローバルなシンボリックリンクを指す)に切り替えてください – これには次を実行して:

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

を行い、次でライブリンクが正しく設定されていることを確認します:

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

`platform-86duino.pio-link` という名前のエントリが表示されるはずです。開いてみると(JSON 形式)、`spec.uri` が解凍済みの `PlatformIO-86Duino-501` フォルダを指しているはずです。

コンパイルエラー: `'cmd_t' has not been declared`(行・列番号は場合による)

→ PIO の `.ino` 用プリプロセッサが、その型を導入するヘッダより前方に前方宣言を挿入してしまいました。修正方法は 2 つあります:

1. **簡易的な方法**: 型を定義しているヘッダを、他の `#include` より前、スケッチの一番上に移動する。
2. **正式な方法**: スケッチを `.ino` から `.cpp` にリネームし、先頭に `#include <Arduino.h>` を追加する – これにより PIO の `.ino` 前処理を完全にバイパスします。

アップロードエラー: `No found the interface on selected board`

→ `platformio.ini` の `board =` の値が誤っているか、ボードの JSON に `build.boardname` フィールドがありません。`platform-86duino\boards\<your_board>.json` を開き、`tool-86duino-uploader\interface.v86` の 1 列目と一致する `"boardname"` フィールドがあることを確認してください(例: `QEC7`、`ONE`、`ZERO`、`EDUCAKE`)。

アップロードエラー: `... Error command` または `Bootloader: Receive a unavailable command`

→ 通常、`platformio.ini` の `board =` の値が接続されているハードウェアと一致していないことを意味します。`pio run -t select-board` を実行して正しい機種を選び直し、再試行してください。ブートローダー／ファームウェアの更新時にはコマンドコードを手動で設定せず、内部で正しい v86dude コマンドを処理する専用ターゲットを使用してください:

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

アップロードエラー: `Open src file: fail, fp = NULL` / アップロードされたファイルが 0 バイト

→ `program.exe` へのパスが解決できませんでした。`pio run` が先に成功していることを確認してください（`.pio/build/<board>/program.exe` が存在している必要があります）。

COM ポートが見つからない

→ 確認事項:

1. USB ケーブルが正しく接続されている（ボードの電源 LED が点灯している）。
2. Windows の **デバイスマネージャー** → **ポート** にデバイスが表示されている。
3. 「不明なデバイス」や黄色い警告マークが表示される場合は、Windows 用 PL2303 ドライバをインストールしてください – QEC ソフトウェアページ（https://www.qec.tw/software/ の「Resources」）からダウンロードできます。

アップロードが進まずハングする

→ ボードが BIOS で止まっている可能性があります。USB を抜き、リセットボタンを押してから再度挿し込み、再アップロードしてください。

VS Code に `C/C++ IntelliSense service does not support .INO files` と表示される

→ これは **VS Code エディタ側の警告であり、PlatformIO のコンパイルやアップロードには一切影響しません。** エディタの IntelliSense（赤い波線・自動補完）が `.ino` ファイルで機能しないことを意味するだけです。

対処方法は 2 つあります:

1. **無視する**: 警告を閉じるだけで問題ありません。
2. **スケッチを `.ino` から `.cpp` に変換する**（IntelliSense を完全に使えるようにする場合）:
3. `src\<sketch>.ino` を `src\<sketch>.cpp` にリネームします。
4. ファイルの先頭に `#include <Arduino.h>` を追加します。
5. `setup()` / `loop()` 以外の場所から呼び出される関数がある場合は、先頭に前方宣言を追加します。

注意: `.cpp` に変換すると、そのスケッチは 86Duino IDE では開けなくなります。両方の IDE を使いたい場合は `.ino` のままにして警告は無視してください。

コマンドチートシート

> リマインダー: `pio run` はビルドのみを行い、自動アップロードは行いません。ボードに書き込むには必ず `-t upload` を使用してください。

```
# 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
```

上級者向け – ビルダー自体をカスタマイズする

ビルドロジック全体は `platform-86duino\builder\main.py`(IDE の `platform.txt` を SCons に翻訳したもの)にあります。コンパイルフラグやリンクするライブラリに誤りを見つけた場合は、このファイルを直接編集してください – **インストールがシンボリックリンクのため、変更は再インストール不要で即座に反映されます。**

ライブリンクがまだ正しく設定されているか確認するには:

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

JSON が表示され、`spec.uri` が解凍済みの `PlatformIO-86Duino-501` フォルダを指していれば問題ありません。

問題が解決しない場合

保守担当者へ、次の 3 つの情報を送ってください:

1. `pio --version` の出力
2. `pio pkg list -g` の出力
3. `pio run -v` の完全なログ — 次のコマンドで取得できます: `pio run -v 2>&1 | Tee-Object -FilePath build.log`

Warranty

This product is warranted to be in good working order for a period of one year from the date of purchase. Should this product fail to be in good working order at any time during this period, we will, at our option, replace or repair it at no additional charge except as set forth in the following terms. This warranty does not apply to products damaged by misuse, modifications, accident or disaster. Vendor assumes no liability for any damages, lost profits, lost savings or any other incidental or consequential damage resulting from the use, misuse of, originality to use this product. Vendor will not be liable for any claim made by any other related party. Return authorization must be obtained from the vendor before returned merchandise will be accepted. Authorization can be obtained by calling or faxing the vendor and requesting a Return Merchandise Authorization (RMA) number. Returned goods should always be accompanied by a clear problem description.

本書に記載されているブランド名および製品名は、各社の所有物および登録商標です。

本書に記載されている名称はすべて、識別目的のみに使用されます。

All Trademarks appearing in this manuscript are registered trademark of their respective owners.
All Specifications are subject to change without notice.

©ICOP Technology Inc. 2025

日本語版資料は、英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。