ESP32-P4

ESP-VISION Programming Guide

Release master
Espressif Systems
Jun 17, 2026

Table of contents

ESP-VISION is a Low-Code Edge AI & Computer Vision Framework for Espressif SoCs. It deeply integrates essential capabilities including camera capture, image processing, video encoding and decoding, network transmission, model deployment, and AI inference, while providing unified and standardized Python APIs that enable developers to rapidly build edge applications combining visual capture, intelligent recognition, display, and media streaming.

This guide is organized as follows.

Chapter 1

Introduction

1.1 What Is ESP-VISION

ESP-VISION is a Low-Code Edge AI & Computer Vision Framework for Espressif SoCs. It deeply integrates essential capabilities including camera capture, image processing, video encoding and decoding, network transmission, model deployment, and AI inference, while providing unified and standardized Python APIs that enable developers to rapidly build edge applications combining visual capture, intelligent recognition, display, and media streaming.

1.2 Key Features

- Unified camera, image, display, video encoding, preview, and streaming APIs across supported chips and boards.
- Image processing capabilities covering drawing, filtering, color tracking, feature detection, QR codes, barcodes, and AprilTags.
- ESP-DL-powered object detection, pose estimation, and image classification, with a streamlined path for deploying models to edge devices.
- Efficient C/C++ foundation components work closely with on-chip multimedia peripherals and hardware acceleration modules to deliver high-performance, real-time application execution.
- Development through a VSCode-based host tool or Web IDE, with firmware builds managed through `idf.py`.

1.3 Supported Boards

ESP-VISION supports boards based on ESP32-P4, ESP32-S3, and ESP32-S31. See *Chip and Board Support* for the full board list and the chip-specific modules and constraints.

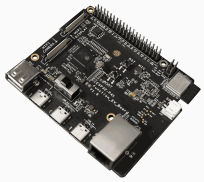
See *Get Started* to build and flash the firmware, and *Solution Architecture* for how the pieces fit together.

Chapter 2

Chip and Board Support

ESP-VISION documentation is built separately for each supported chip. Select the chip used by the board firmware so that the documented modules and APIs match the default build.

2.1 Supported Development Boards

Picture	Board	Chip	ESP-VISION support
	ESP32-P4X-EYE	`ESP32-P4`	**Supported** | Modules: `sensor`, `image`, `display`, `espdl`, `imageio`, `h264`, `rtsp`, `barcode` | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-Function-EV-Board	`ESP32-P4`	**Supported** | Modules: `sensor`, `image`, `display`, `espdl`, `imageio`, `h264`, `rtsp`, `barcode` | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-S3-EYE	`ESP32-S3`	**Supported** | Modules: `sensor`, `image`, `display`, `espdl`, `imageio` | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-S31-Korvo	`ESP32-S3`	**Supported (ESP-IDF master only)** | Modules: `sensor`, `image`, `display`, `espdl`, `imageio` | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag

Note: The support summary is generated from `mpconfigport.h`, each board `mpconfigboard.h` and `imlib_config.h`, and the chip-specific module selection.

2.2 Chip Versus Board Capabilities

The selected chip controls chip-level source selection. For example, `h264` and `rtsp` are added by `micropython.cmake` only when `IDF_TARGET` is `esp32p4`. A board profile can further enable or disable features such as the ZXing-C++ barcode backend through `boards/<BOARD>/board.cmake`.

The API navigation and chip-specific symbol conditions are generated from these build files. This makes `micropython.cmake`, `boards/*/port/mpconfigboard.cmake`, and board `board.cmake` files authoritative for ESP-VISION API availability.

2.3 Standard MicroPython Features

Standard MicroPython features use a separate configuration path: `overlay/micropython/ports/esp32/mpconfigport.h` plus each board's `mpconfigboard.h`. The table above indexes the explicitly enabled baseline features from those files, while additional conditions can depend on chip capabilities and firmware configuration.

The current board headers disable Bluetooth and ESP-NOW; the S31 board additionally disables `machine.ADC` and `machine.ADCBlock`. A board must still provide suitable networking hardware and firmware configuration before `network.WLAN` can connect.

Because these conditions are not determined solely by the chip, the chip selector filters the ESP-VISION API reference but does not represent a complete standard MicroPython module manifest. Build-environment compatibility details are maintained in the project README.

The chip documentation describes the defaults shared by the current board profiles. A custom board or modified build option can therefore differ from the published chip page. When that happens, the firmware configuration is authoritative.

Chapter 3

Get Started

3.1 Prerequisites

- A supported ESP-IDF environment with its export script sourced so that `idf.py` is on `PATH`; see the project README for the maintained branch compatibility.
- A board listed in *Chip and Board Support* for the selected chip.

3.2 Build, Flash, and Monitor

Clone the repository with submodules, then use the board-aware `idf.py` extension from the repository root:

```
git clone --recursive https://github.com/espressif/esp-vision.git esp-vision
cd esp-vision
```

```
idf.py --board ESP32_P4X_EYE -p /dev/ttyACM0 build flash monitor
```

The command first runs `prepare-micropython`: it verifies that `lib/micropython` is checked out at the pinned MicroPython v1.28.0 commit, exports a clean MicroPython build copy under `build/micropython/`, applies `overlay/micropython/` to that copy, then projects each `boards/<BOARD>/port/` onto the copy's `ports/esp32/boards/<BOARD>/`. `lib/micropython` is never dirtied.

3.3 Common idf.py Commands

Command	Description
`idf.py --board <BOARD> build`	Build firmware for a board.
`idf.py --board <BOARD> -p <PORT> flash`	Build and flash firmware.
`idf.py --board <BOARD> -p <PORT> monitor`	Open the serial monitor.
`idf.py --board <BOARD> menuconfig`	Open menuconfig.
`idf.py --board <BOARD> -p <PORT> erase-flash`	Erase flash.
`idf.py --board <BOARD> clean`	Clean board build output.
`idf.py --board <BOARD> full-clean`	Remove the complete build directory for the selected board.

3.4 Run Your First Script

After flashing, connect over the REPL and try the camera. Each *API Reference* module page links the runnable `example/` scripts for that API.

Chapter 4

Concepts

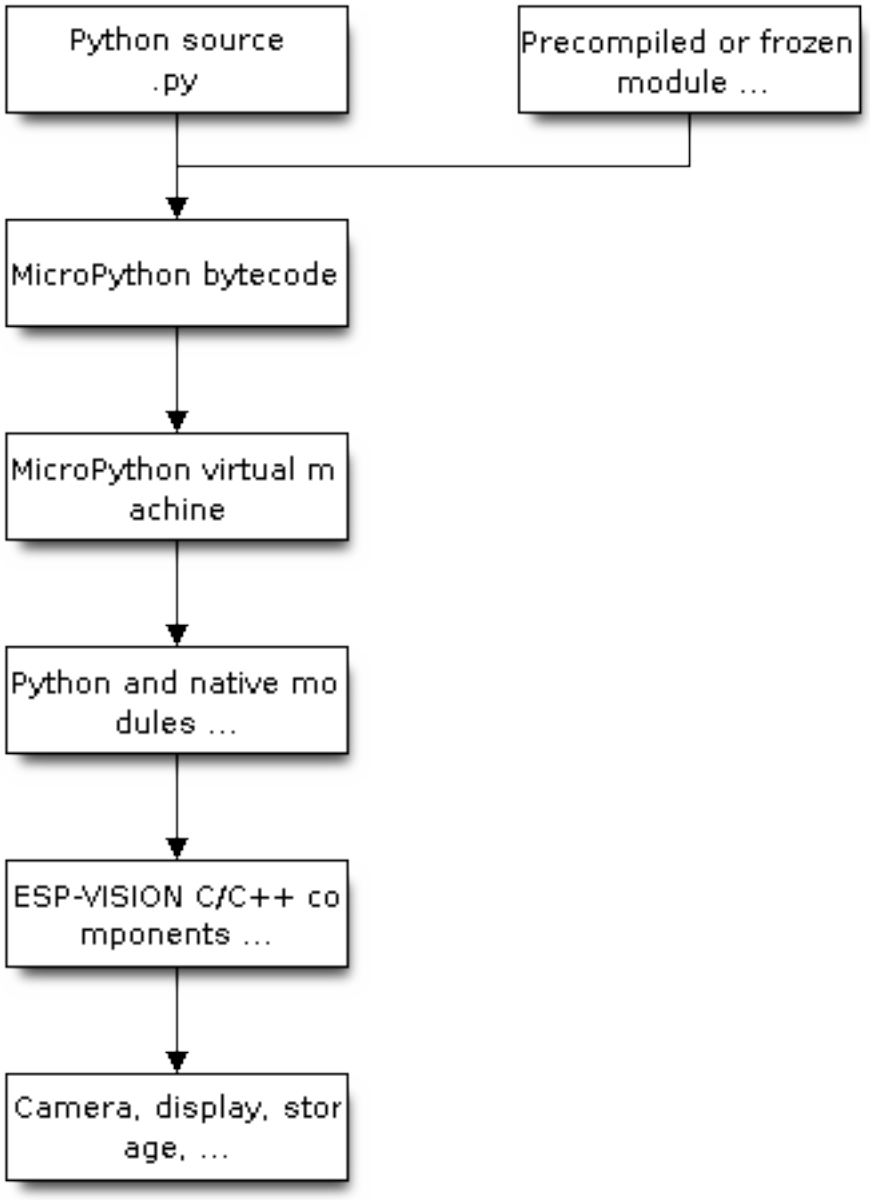
This section explains the ideas behind the ESP-VISION API: how images are represented in memory, how the image-processing algorithms work, how neural network inference runs on the device, and how frames move through the camera and codec pipelines. Read it alongside the *API Reference* to understand *why* each function behaves the way it does.

4.1 MicroPython Language Fundamentals

MicroPython is a compact implementation of the Python language designed to run directly on microcontrollers. ESP-VISION uses MicroPython v1.28.0 as its application language while retaining C/C++ and ESP-IDF components for hardware access and performance-critical vision work. This chapter introduces the language model needed to read and write ESP-VISION applications; the authoritative details remain in the MicroPython v1.28.0 language and implementation reference.

4.1.1 Execution Model

Upstream MicroPython aims to implement Python 3.4 syntax with selected features from later Python versions. ESP-VISION builds the ESP32 port at the `EXTRA_FEATURES` configuration level, with the on-device compiler, garbage collector, MPZ long integers, single-precision floating point, scheduler, VFS, and persistent bytecode loading enabled. Source code remains dynamically typed Python, but it is compiled to compact bytecode and executed by the MicroPython virtual machine. Native modules connect Python objects to ESP-VISION C/C++ components, ESP-IDF drivers, and chip hardware:



Code loaded from `.py` is compiled on the device and consumes RAM during compilation. Precompiled `.mpy` files and modules frozen into the firmware reduce runtime compilation work; immutable constants in frozen modules can remain in Flash. Python still owns application policy and object lifetimes, while native components implement operations such as frame capture, image processing, codecs, and AI inference.

4.1.2 Basic Syntax

The current ESP-VISION configuration uses indentation to define blocks and supports the familiar Python forms for assignment, expressions, conditions, loops, functions, classes, exceptions, context managers, comprehensions, generators, imports, and `async`/`await`. Names are dynamically bound to objects, so a declaration does not specify a C-style storage type:

```
from micropython import const

_MIN_PIXELS = const(80)

class Detection:
    def __init__(self, label, score):
        self.label = label
        self.score = score

    def accepted(self, threshold=0.5):
        return self.score >= threshold

def select_results(results):
    selected = []
    for label, score in results:
        item = Detection(label, score)
        if item.accepted():
            selected.append(item)
    return selected

try:
    detections = select_results((("person", 0.91), ("chair", 0.42)))
    for item in detections:
        print(item.label, item.score)
except (ValueError, TypeError) as error:
    print("invalid result:", error)
finally:
    print("processing complete")
```

`const()` is a MicroPython extension that lets the compiler substitute a constant value and can reduce bytecode and global-dictionary overhead. Type annotations may be useful to developers and editors, but MicroPython does not use them to provide C-style static typing or automatic memory layout.

4.1.3 Main Data Structures

The choice between mutable and immutable objects matters on a microcontroller because creating, resizing, or concatenating objects allocates heap memory:

Type	Mutability	ESP-VISION usage
`None` and `bool`	Immutable	Represent absence and state flags without defining custom sentinel objects.
`int` and `float`	Immutable	Store counters, coordinates, thresholds, and scores. Repeated arithmetic can create new objects; avoid floating-point work in hard interrupt context.
`str`	Immutable	Store text, paths, labels, and JSON keys. Concatenation creates a new string, so avoid building large strings every frame.
`bytes`	Immutable	Store compact binary constants, encoded packets, and read-only model or protocol data.
`bytearray`	Mutable	Provide reusable binary buffers for peripheral I/O and packet assembly without allocating a new object for every operation.
`list`	Mutable	Store ordered results whose size changes. `append()` and growth may allocate memory, so bound or reuse lists in sustained frame loops.
`tuple`	Immutable	Represent fixed records such as rectangles and detection results. Constant tuples can be more memory-efficient than repeatedly created lists.
`dict`	Mutable	Represent configuration and shared scalar state. Adding keys can resize the table; create the expected structure during initialization.
`set`	Mutable	Perform membership and uniqueness checks when the additional hash-table memory is justified.
`range`	Immutable	Describe integer iteration without constructing a list of all values.
`memoryview` and `array`	View / mutable	Access typed or sliced buffer data with fewer copies; the referenced buffer must remain alive and must not be resized while a view is active.

Object references are shared rather than copied automatically. Assigning `b = a` makes both names refer to the same mutable list, dictionary, image, or buffer; use an explicit copy only when independent data is required. This is especially important for *`image.Image`* objects backed by reusable camera frame buffers.

4.1.4 Design Principles for Microcontrollers

MicroPython prioritizes portability, interactive development, compact firmware integration, and direct hardware access over complete CPython library coverage. Its garbage-collected heap makes dynamic application code practical, but RAM, Flash, stack space, and allocation latency remain finite. The MicroPython guide for constrained devices recommends reducing repeated object creation, preallocating buffers, moving constants and reusable modules into frozen bytecode, and monitoring the heap with `gc.mem_free()` and `gc.mem_alloc()`.

For ESP-VISION applications, initialize modules, models, buffers, and stable dictionaries before entering the frame loop. Reuse `bytearray` and image buffers, keep per-frame temporary data bounded, and avoid retaining a camera-backed image after the next `sensor.snapshot()` unless it has been copied. Use Python to express orchestration and product behavior; leave deterministic real-time work, ISR handling, bulk pixel processing, codecs, and inference kernels in native components.

4.1.5 MicroPython and CPython

In this documentation, "Python" in a runtime comparison means CPython 3.x, the reference implementation commonly used on PCs and servers. The exact MicroPython feature set is firmware-dependent, so this table describes MicroPython v1.28.0 as configured by ESP-VISION: all maintained board manifests freeze `asyncio`, the ESP32 port enables `_thread` with a GIL, and hardware APIs remain subject to chip and board configuration.

Area	ESP-VISION MicroPython	CPython
Primary environment	Runs inside MCU firmware with direct access to peripherals and board services.	Runs as an operating-system process on PCs, servers, and larger embedded systems.
Language syntax	Implements Python 3.4 syntax with selected later features; most common control flow, functions, classes, exceptions, generators, and asynchronous syntax are available.	Tracks the current Python language specification and generally introduces new syntax first.
Type system	Dynamically typed; annotations do not make execution statically typed.	Dynamically typed; annotations are consumed mainly by tools and optional libraries.
Standard library	Provides a resource-oriented subset selected at firmware build time, plus `machine`, `micropython`, `esp`, and `esp32` hardware/runtime modules.	Provides the full CPython standard library for the installed version, without a standard MCU peripheral API.
Package deployment	Modules are copied to storage, precompiled as `.mpy`, or frozen into firmware; desktop `pip` and arbitrary binary wheels must not be assumed.	Commonly uses `pip`, virtual environments, source distributions, and platform wheels.
Memory model	Uses a small garbage-collected heap shared with firmware resources; allocation failures and fragmentation must be considered explicitly.	Uses a much larger process heap and reference counting with cyclic garbage collection.
Concurrency	Provides frozen `asyncio`, enabled `_thread` with a GIL, and scheduled GPIO IRQ and Timer callbacks; peripheral availability still depends on the selected chip and board.	Provides `asyncio`, threads, processes, and extensive operating-system synchronization facilities.
Performance path	Calls native C/C++ modules and hardware accelerators for demanding operations; Python is best used for orchestration.	Can use optimized extension modules, JIT-enabled alternative runtimes, or multiprocessing, but usually has greater CPU and memory resources.
Compatibility details	Some built-in behavior, exception text, introspection, module contents, and edge cases differ to reduce code size and RAM usage.	Defines the reference behavior against which MicroPython differences are documented.
Portability	Pure Python code using the supported subset is often portable; hardware modules and resource assumptions are device-specific.	Pure Python code is portable across supported operating systems when its dependencies are available.

Do not decide portability from syntax alone. Check imported modules, memory use, filesystem paths, concurrency assumptions, and hardware access. The upstream MicroPython and CPython differences list documents known behavioral differences, while *MicroPython APIs* describes the inherited APIs and concurrency model available in ESP-VISION.

4.2 Packages and Deployment

MicroPython organizes reusable code as modules and packages, but an embedded product must also decide where that code is stored and how it is updated. This chapter describes package management and deployment for the MicroPython v1.28.0 version pinned by ESP-VISION; see the upstream package management and manifest references for the complete generic behavior.

4.2.1 Module and Package Layout

A module is normally a `.py` source file or a precompiled `.mpy` file. A package is a directory containing an `__init__.py` file and one or more modules or subpackages, so a reusable vision pipeline can use the following layout:

```
my_vision/
    __init__.py
    pipeline.py
    transport.py
```

Applications import package members with standard Python syntax:

```
from my_vision.pipeline import VisionPipeline

pipeline = VisionPipeline()
```

4.2.2 Package Management in the Current Firmware

MicroPython uses `mip` rather than CPython's `pip` to install packages from micropython-lib, package indexes, URLs, or a local `package.json` description. The default ESP-VISION manifests do not currently embed the device-side `mip` module, so `import mip` is not available in the standard firmware; use `mpremote` on the development host to install packages into the device filesystem:

Important: The default ESP-VISION firmware does not support device-side `mip`. Running `import mip` or `mip.install()` on the device raises `ImportError`. The host-side `mpremote mip` command remains available and is the default package installation method.

```
mpremote connect <PORT> mip install --target=/lib <PACKAGE>
mpremote connect <PORT> mip install --target=/lib <PACKAGE>@<VERSION>
mpremote connect <PORT> mip install --target=/lib ./package.json
```

A `package.json` file describes the files and dependencies that `mip` installs at runtime or during provisioning. It is not a firmware manifest and does not determine which modules are compiled into ESP-VISION. Package compatibility must be checked against MicroPython v1.28.0 and the selected chip architecture because MicroPython packages can depend on firmware features or architecture-specific `.mpy` files.

Device-side installation with `mip.install()` requires a customized firmware that explicitly includes `mip` together with the required networking and TLS support. Host-side installation with `mpremote` is the recommended default because it does not increase the product firmware image or require the device to download code from the network.

4.2.3 Packages on the Filesystem

ESP-VISION mounts the internal Flash filesystem at `/` and adds `/lib` to `sys.path`. Install reusable packages under `/lib`; modules in the current directory, frozen modules, and packages under `/lib` can then be imported without changing the application code.

An SD card is mounted at `/sdcard` when enabled, but its package directory is not added to `sys.path` automatically. Add it explicitly before importing packages stored there:

```
import sys

sys.path.append("/sdcard/lib")
from my_vision.pipeline import VisionPipeline
```

Source `.py` files are compiled when imported, which consumes RAM for bytecode and adds import latency. Pre-compiled `.mpy` files avoid compilation on the device and reduce source exposure, but their bytecode still occupies RAM after loading and they must match the MicroPython version and architecture.

4.2.4 Packages Embedded in Firmware

Firmware manifests select Python modules and packages that are compiled and frozen into the firmware image. ESP-VISION keeps board manifests in `boards/<BOARD>/manifest.py` and currently uses them to freeze the port modules, ESP-VISION initialization modules, `asyncio`, and selected board-specific features.

For example, place `my_vision/__init__.py` and its modules under `boards/<BOARD>/packages`, then add the package to that board's manifest:

```
package(
    "my_vision",
    base_path="$(ESP_VISION_ROOT)/boards/<BOARD>/packages",
)
```

The lower-level `freeze()` function can also freeze a directory or selected modules, while `include()` and `require()` reuse manifest fragments and packages from configured manifest libraries. Reconfigure and rebuild the selected board after changing its manifest:

```
idf.py --board <BOARD> reconfigure
idf.py --board <BOARD> build
idf.py --board <BOARD> flash
```

4.2.5 Choosing a Deployment Method

Use the expected update frequency and product ownership of a package to select its deployment method:

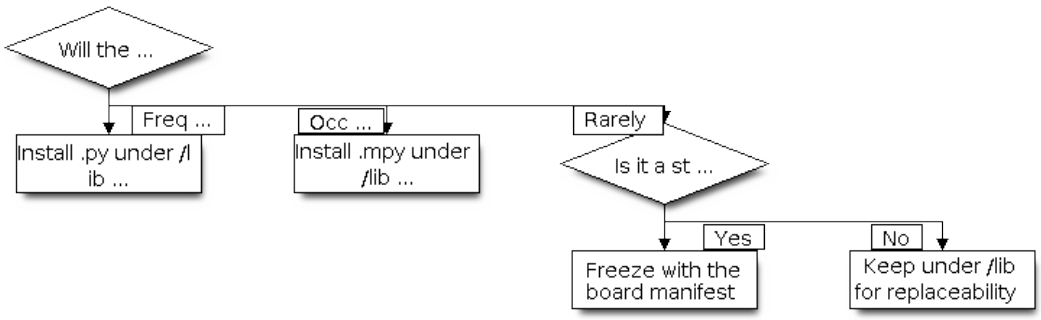


Fig. 1: Package deployment decision

Table 1: Deployment method comparison

Property	`.py` under `/lib`	`.mpy` under `/lib`	Frozen in firmware
Storage	Writable filesystem	Writable filesystem	Firmware Flash image
Import behavior	Compiled when imported	Loaded without source compilation	Executes directly from frozen bytecode
Runtime RAM	Bytecode and objects use RAM	Bytecode and objects use RAM	Frozen bytecode and constants can remain in Flash
Update method	Replace files or use `mpremote mip`	Replace compatible `.mpy` files	Rebuild and flash firmware
Primary use	Development and frequently updated application code	Replaceable modules with lower import overhead	Stable product dependencies and startup modules

4.2.6 Import Priority and Version Control

The current firmware searches the current directory, frozen modules through `.frozen`, and then `/lib`. A filesystem package with the same import name therefore does not override a frozen package during normal import; remove the frozen package from the manifest and rebuild the firmware, or use a different package name.

Keep board manifests and frozen package revisions in source control so firmware builds are reproducible. Use `package.json` to describe files and dependencies installed by `mip`, and use `manifest.py` to define modules embedded during the firmware build; these formats serve different deployment stages and are not interchangeable.

For ESP-VISION products, freeze stable framework extensions and startup dependencies, while keeping product scripts, configuration, and field-updatable modules under `/lib`. Large AI models, images, and video assets should normally remain data files in Flash, SD card, or external storage instead of being embedded as Python package data, so they can be updated and mapped by the appropriate runtime component.

4.3 The Image Model

Every vision operation in ESP-VISION reads or writes an *`image.Image`*: a rectangular grid of pixels plus a small header describing its width, height, and pixel format. Understanding how those pixels are laid out, which color spaces the algorithms use, and how the underlying memory is managed is the key to writing fast, correct scripts.

4.3.1 Pixel formats

The pixel format determines how many bytes each pixel occupies and how its value is interpreted.

Format	Bytes/pixel	Description
`BINARY`	1 bit	Black/white. Used for masks and the output of *`image.Image.binary()`*.
`GRAYSCALE`	1	8-bit intensity (0-255). The format most filters and detectors prefer.
`RGB565`	2	16-bit color: 5 bits red, 6 bits green, 5 bits blue.
`BAYER`	1	Raw single-channel mosaic straight from the sensor; debayered on demand.
`YUV422`	2	Luminance/chrominance packing, common in camera/codec pipelines.
`JPEG` / `PNG`	varies	Compressed byte streams produced by *`image.Image.to_jpeg()`* / `to_png`.

RGB565 packs a color into 16 bits, trading color precision for half the memory of RGB888. Green gets the extra bit because the human eye is most sensitive to it. *`image.Image.get_pixel()`* can return either the packed value or an `(r, g, b)` tuple expanded back to 8 bits per channel.

4.3.2 Color spaces

Although pixels are *stored* as grayscale or RGB565, color thresholding works in the **LAB** color space. LAB separates lightness (L) from two color-opponent axes (A: green-red, B: blue-yellow), so a threshold expressed in LAB is far more robust to brightness changes than one in RGB. This is why a color threshold is a six-tuple `(l_min, l_max, a_min, a_max, b_min, b_max)` while a grayscale threshold is just `(min, max)`.

The module-level conversion helpers (*`image.rgb_to_lab()`*, *`image.lab_to_rgb()`*, *`image.rgb_to_grayscale()`*, and the rest of the `*_to_*` family) expose these conversions for a single pixel, which is handy when computing thresholds offline.

4.3.3 The frame buffer and `fb_alloc`

Embedded memory is scarce, so ESP-VISION avoids per-frame heap allocation. *`sensor.snapshot()`* returns an image backed by a reusable **frame buffer**: the next `snapshot()` overwrites it. If you need to keep a frame around (for example, to compare against a later frame in *`image.Image.difference()`*), make an explicit `img.copy()`.

Many algorithms need scratch memory that lives only for the duration of a call. These use a stack-like allocator, `fb_alloc`, that carves temporary buffers out of the frame-buffer region and frees them all at once when the operation returns. This is why heavy methods do not fragment the heap, and why you should prefer reusing one image over allocating new ones in a hot loop.

4.3.4 Region of interest (ROI)

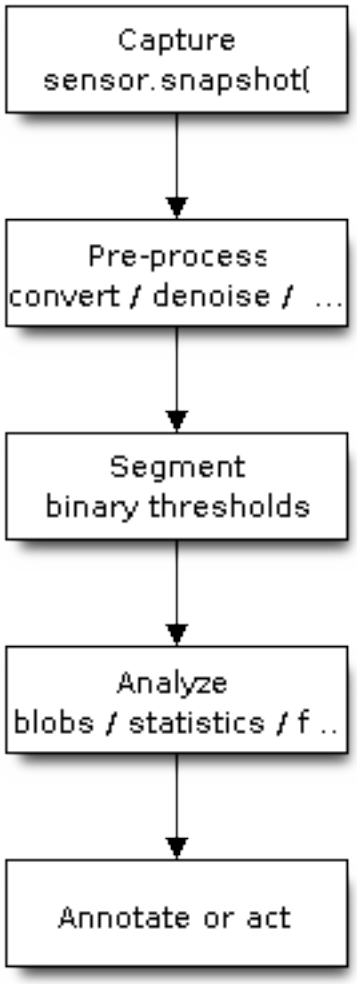
Most analysis and conversion methods accept an `roi=(x, y, w, h)` keyword that restricts the operation to a sub-rectangle of the image. Working on an ROI is both faster (fewer pixels) and more selective (ignore irrelevant areas). The ROI is always expressed in the source image's pixel coordinates.

4.4 Image Processing

The vision algorithms in the *image –Image Processing* module come from OpenMV's `imlib` library, maintained in this repository as the `components/imlib` IDF component. They fall into a few families: point and geometric transforms, neighborhood filters, statistical analysis, and feature detection. This page explains what each family does and when to reach for it.

4.4.1 A typical processing pipeline

Most color-tracking or inspection scripts follow this processing flow:



Pre-processing can combine `image.Image.to_grayscale()`, `image.Image.gaussian()`, and `image.Image.histeq()`; segmentation commonly uses `image.Image.binary()`; analysis then uses `image.Image.find_blobs()`, `image.Image.get_statistics()`, or a feature detector before the application draws results or takes an action.

4.4.2 Thresholding and segmentation

`image.Image.binary()` converts the image into a mask by testing each pixel against a list of color thresholds (see *The Image Model* for why thresholds are expressed in LAB). A pixel is "on" if it falls inside *any* of the supplied thresholds, which lets you track several colors at once. The companion `image.Image.find_blobs()` runs an 8-connected component labeling pass over the same thresholds and returns one `image.blob` per connected region, with centroid, bounding box, pixel count, orientation, and shape descriptors such as roundness and solidity.

4.4.3 Neighborhood (convolution) filters

These replace each pixel with a function of its `(2*ksize+1)` square neighborhood:

- `image.Image.mean()` –box average; fast blur.

- *image.Image.gaussian()* –weighted average using a Pascal-triangle (binomial) approximation of a Gaussian; the standard noise-reduction blur. Set `unsharp=True` to sharpen instead.
- *image.Image.laplacian()* –second-derivative edge response; set `sharpen=True` to add it back to the original for edge enhancement.
- *image.Image.morph()* –apply an arbitrary integer kernel. `mul` and `add` scale and bias the result; the default `mul` is `1/sum(kernel)` so the image brightness is preserved.

All of these share `threshold`/`offset`/`invert` keywords that turn the filter into an adaptive thresholding operation in a single pass.

4.4.4 Rank and edge-preserving filters

Rank filters sort the neighborhood instead of averaging it, which removes noise without blurring edges as much:

- *image.Image.median()* –the percentile (default 0.5) value; excellent against salt-and-pepper noise.
- *image.Image.mode()* –the most common value.
- *image.Image.midpoint()* –a `bias` blend between the min and max.
- *image.Image.bilateral()* –a Gaussian blur weighted by both spatial distance (`space_sigma`) and color similarity (`color_sigma`), so it smooths flat regions while keeping edges crisp.

4.4.5 Morphology

image.Image.erode() and *image.Image.dilate()* grow or shrink the set pixels of a (usually binary) image using a square structuring element; *image.Image.open()* and *image.Image.close()* combine them to remove specks or fill small holes. The `threshold` argument controls how many neighbors must be set, generalizing the classic binary operators.

4.4.6 Histograms and statistics

image.Image.get_histogram() bins pixel values per channel; the returned *image.histogram* can compute Otsu thresholds (*image.histogram.get_threshold()*), percentiles, and full statistics. *image.Image.get_statistics()* returns mean, median, mode, standard deviation, quartiles, and min/max in one call, which is useful for auto-tuning thresholds at runtime.

4.4.7 Feature detection

Higher-level detectors find geometric structure:

- *image.Image.find_lines()* and *image.Image.find_circles()* use the Hough transform; their `threshold` is the minimum accumulator score and the `*_margin` keywords merge near-duplicate results.
- *image.Image.find_rects()* locates quadrilaterals (useful for fiducials and screens).
- *image.Image.find_qrcodes()* and *image.Image.find_apriltags()* detect and decode the corresponding marker types; AprilTags can additionally return 6-DoF pose when camera intrinsics are supplied.

The current ESP32-P4 board profiles additionally provide *image.Image.find_barcodes()* through the ZXing-C++ backend.

Note: GPL-licensed OpenMV code paths are disabled in this build (`OMV_NO_GPL=1`), so a small number of upstream algorithms are intentionally unavailable. The per-board `imlib_config.h` selects which optional algorithms are compiled in; a method that is not enabled raises an exception at call time.

4.5 AI Inference

The *espdl –Model Inference* module runs neural networks on the device using ESP-DL, Espressif's deep-learning inference library. ESP-VISION wraps ESP-DL in task-specific classes so a script can go from an image to detections, poses, or class labels in a few lines, while ESP-DL handles the heavy tensor math on the chip's vector unit (and, on ESP32-P4, additional acceleration).

4.5.1 The `.espdl` model format

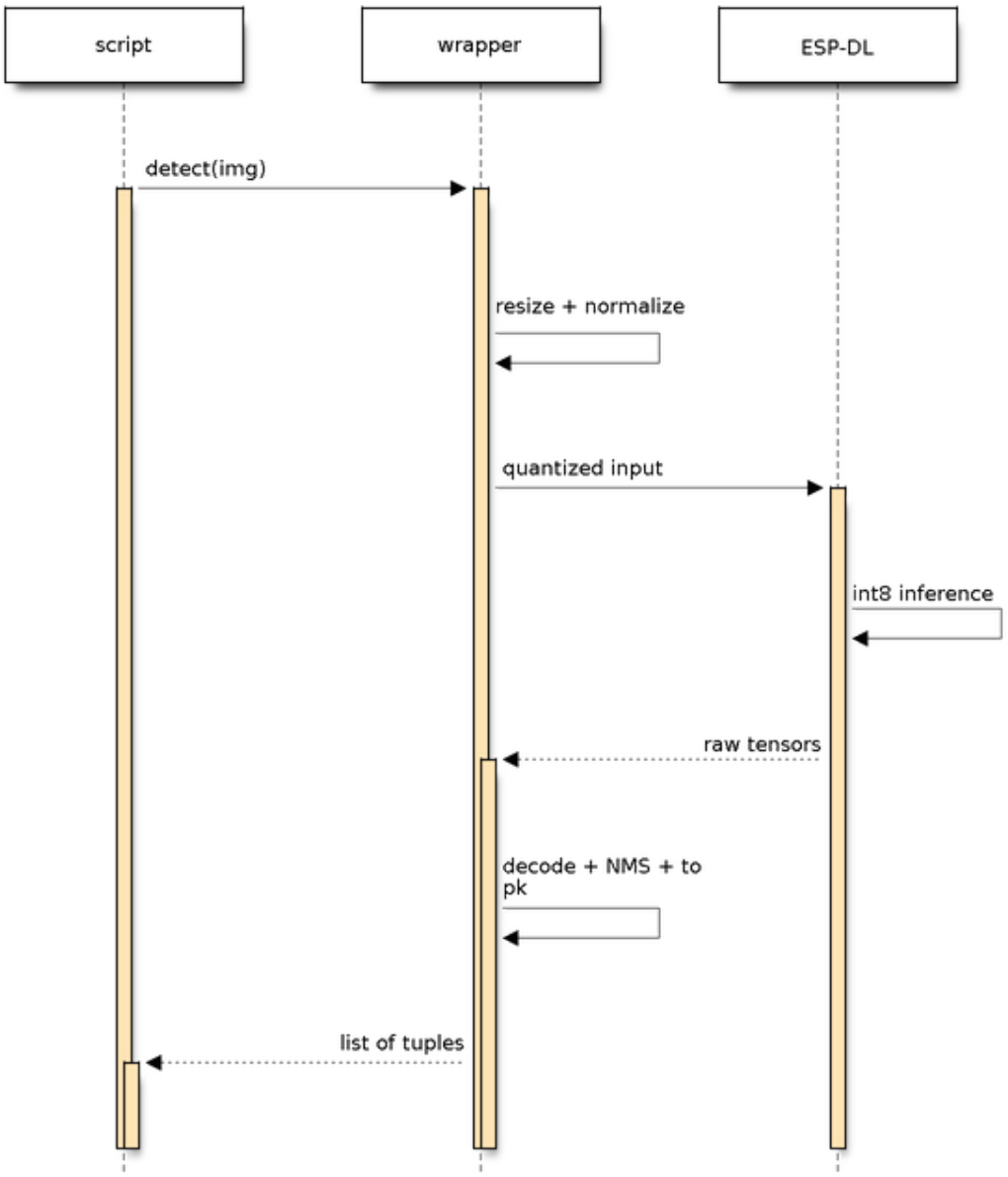
Models are distributed as `.espdl` files. An `.espdl` bundle contains the network graph, the trained weights, and – crucially –**quantization** information. A model is converted from a floating-point framework (PyTorch, TensorFlow) into ESP-DL format ahead of time; see *Introduce a New Model* for the deployment workflow. Models live on storage (`/sdcard/...`) or the flash data partition (`/flash/...`) and are loaded with the wrapper constructor or *`espdl.load_model()`*.

4.5.2 Quantization

Microcontrollers have no FPU budget for full 32-bit float inference at frame rate, so ESP-DL runs **quantized** models, typically 8-bit (and 16-bit where accuracy demands it) integer arithmetic. Quantization maps the float range of each tensor onto integers with a scale factor, shrinking the model and letting the hardware multiply-accumulate integers far faster. The accuracy cost is usually small, but it is the reason a model must be quantized during conversion rather than loaded directly from its training checkpoint.

4.5.3 Inference flow

A single `detect()` (or `classify()`) call hides several stages. The wrapper prepares the input, ESP-DL runs the quantized network, and the wrapper decodes the raw output into Python tuples:



4.5.4 Pre-processing

Before inference the input image must match what the model was trained on:

- **Resize** to the model's input resolution. The wrappers handle this, scaling the captured frame to the network input size.
- **Color** order and channel count must match (most detectors expect RGB).
- **Normalization** subtracts a per-channel `mean` and divides by a per-channel `std`. These are constructor keywords (`mean=`/`std=`) so you can match the exact preprocessing the model expects; the defaults suit the bundled models.

4.5.5 Post-processing

The raw network outputs are decoded into friendly Python tuples:

- **Object detection** (*espdl.ESPDet*, *espdl.YOLO11*) produces candidate boxes with class scores. A confidence `score` threshold drops weak boxes and **non-maximum suppression** (`nms`) removes overlapping duplicates, leaving `(x, y, w, h, score, category)` tuples. `YOLO11` also caps results with `topk`.
- **Pose estimation** (*espdl.YOLO11nPose*) adds 17 COCO keypoints per detection.
- **Classification** (*espdl.ImageNetCls*) applies an optional `softmax` and returns the `topk (label, score)` pairs.

Thresholds can be tuned at runtime with `set_thresholds` without reloading the model, which is handy when adapting to lighting or distance.

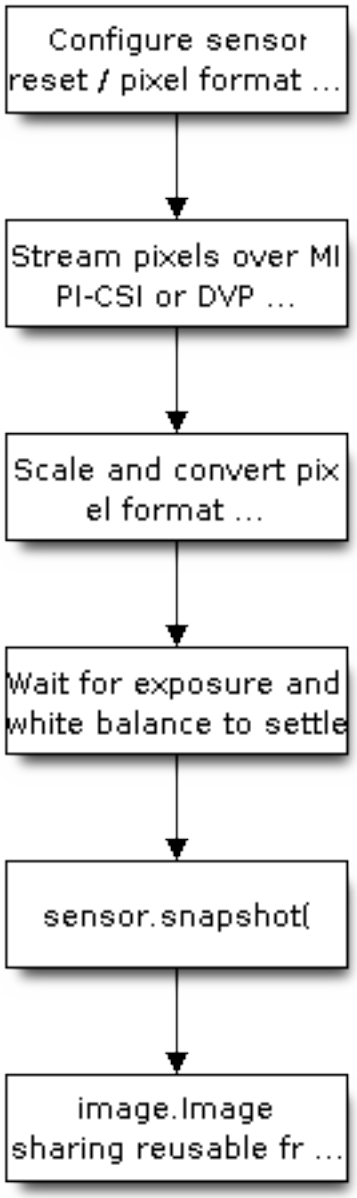
4.5.6 Memory and performance

Models and their activation buffers are large and are allocated in PSRAM. Load a model once and reuse the wrapper across frames rather than recreating it per frame. Call `deinit()` (or drop the last reference) to free a model when you are done. Detection cost scales with input resolution, so smaller models such as `espdet_pico` run at higher frame rates than full-size networks.

4.6 The Camera Pipeline

sensor.snapshot() looks like a single call, but behind it sits a hardware-specific capture pipeline that turns photons into an *image.Image*. Knowing the stages helps explain the configuration calls (`set_pixformat`, `set_framesize`, `skip_frames`) and the performance characteristics of each board.

4.6.1 From sensor to image



The sensor control bus applies the board defaults and requested format before streaming begins. Double buffering allows the next frame to fill while the current frame is processed. ESP32-P4 uses the hardware PPA for scaling and color conversion, while ESP32-S3 performs conversion in software. Calling `sensor.skip_frames(time=2000)` after configuration gives automatic exposure and white balance time to converge.

4.6.2 Per-board backends

The camera backend is selected per board at build time:

Board family	Backend	Notes
ESP32-P4	`esp_video` / V4L2 + PPA	MIPI-CSI sensors; hardware-accelerated scaling and color conversion.
ESP32-S3	`esp32-camera`	DVP sensors; conversion in software, so prefer smaller frame sizes.

Because the public `sensor` API is identical across boards, the same script runs on both; only the achievable resolution and frame rate differ.

4.6.3 Frame sizes

`set_framesize` accepts named sizes (`QQVGA`, `QVGA`, `VGA`, ⋯) drawn from a shared framesize table. Smaller frames cost less memory and less processing time per stage, so when a model or algorithm only needs a small input, capture small rather than capturing large and downscaling.

4.7 Codecs and Streaming

Once you have a frame you often want to *get it off the board*: preview it on a host, record it, or stream it live. ESP-VISION offers several codecs and transports, each suited to a different use case.

4.7.1 Still-image codecs: JPEG and PNG

`image.Image.to_jpeg()` (and its alias *`image.Image.compress()`*) encodes a frame to JPEG. JPEG is lossy: the `quality` keyword (1-100) trades size for fidelity, and `subsampling` controls how aggressively chroma is reduced (`JPEG_SUBSAMPLING_420` is smallest, `444` is best quality). On boards with a hardware JPEG encoder this is offloaded; otherwise a software encoder (`esp_new_jpeg`) is used. *`image.Image.to_png()`* produces lossless PNG, which is larger but exact –useful for saving reference frames.

4.7.2 Recording sequences: ImageIO

`imageio.ImageIO` records a *sequence* of frames with their inter-frame timing preserved, so playback runs at the original speed. A **file stream** writes a container on storage; a **memory stream** keeps frames in a pre-allocated PSRAM buffer for fast capture and replay. See *image.ImageIO –Image Stream*.

4.7.3 Video codec: H.264

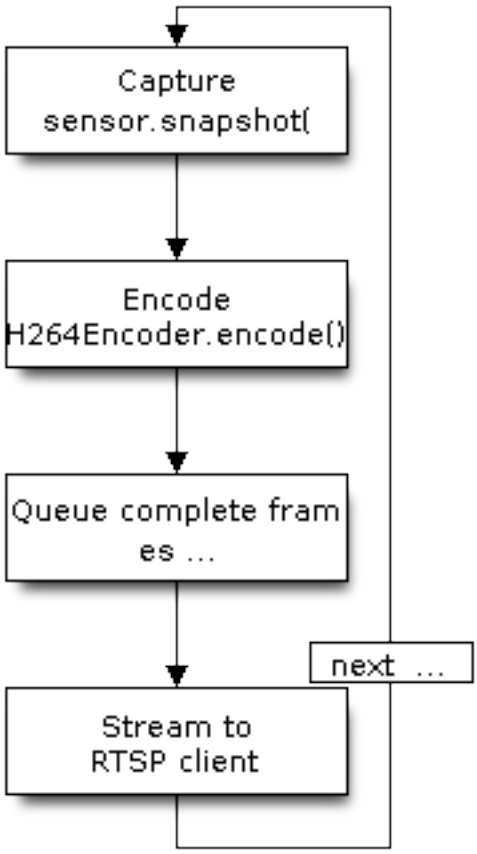
For continuous video, per-frame JPEG is wasteful because it cannot exploit the similarity between consecutive frames. *`h264.H264Encoder`* produces an H.264 stream where occasional **keyframes** (IDR/I) are self-contained and the frames between them only encode what changed. The encoder's parameters tune this trade-off:

- `gop` –keyframe interval. Shorter means faster recovery and easier seeking but larger output; `0` selects one keyframe per second.
- `bitrate` –target bits per second; the dominant size/quality control.
- `qp_min` / `qp_max` –bound the per-frame quantizer.

A lost P-frame corrupts the picture until the next keyframe, which matters for the streaming transport below.

4.7.4 Network streaming: RTSP

rtsp.RTSPServer serves encoded H.264 frames over RTSP so a client such as VLC or ffplay can view the camera at `rtsp://<board-ip>:8554/`. Feed each *h264.H264Encoder.encode()* result to *rtsp.RTSPServer.send()*.



The server keeps a short, in-order frame queue; if a client is slow or absent it drops whole frames rather than blocking the capture loop or sending a partial frame that would corrupt the stream.

4.7.5 Host preview: USB CDC

During development the quickest way to *see* what the camera sees is *image.Image.flush()*, which pushes the current frame to the host as an EVFRAME JPEG preview over USB CDC. The companion host tool decodes and displays it. This path needs no Wi-Fi and no SD card, making it the default feedback loop while iterating on a script.

4.8 Startup Sequence

ESP-VISION follows the MicroPython reset and boot model while adding initialization for the Flash filesystem, board storage, camera, display, and preview services. This chapter describes the sequence implemented by the current firmware. For the upstream model and general behavior, see the MicroPython v1.28.0 Reset and Boot Sequence.

4.8.1 Hard Reset and Soft Reset

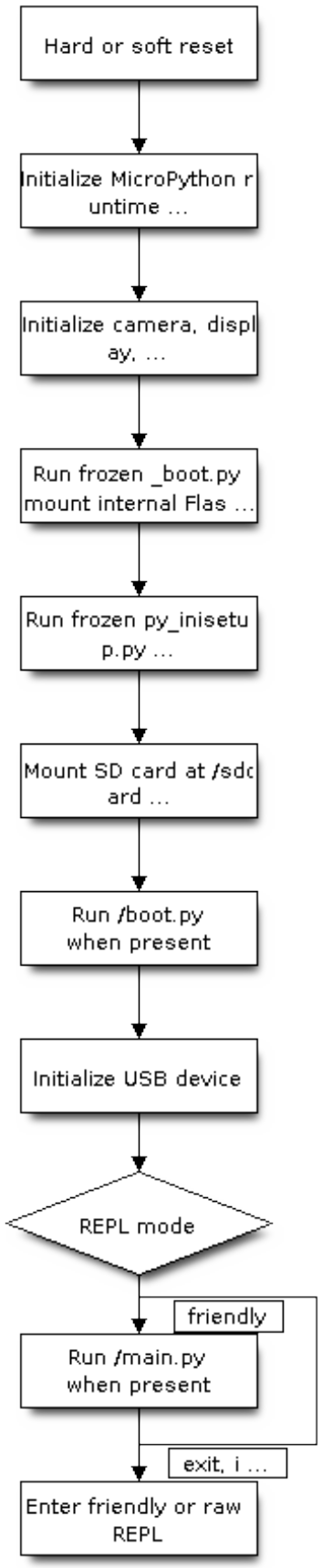
A hard reset restarts the MCU and ESP-IDF runtime before creating a new MicroPython environment. It occurs after power-on, the board reset button, `machine.reset()`, watchdog or brownout reset, and wake-up from deep sleep. Use `machine.reset_cause()` when an application needs to distinguish the reset source.

A soft reset restarts the MicroPython environment without restarting the complete MCU runtime. It can be requested with `Ctrl-D` in the friendly REPL or `machine.soft_reset()`. ESP-VISION clears Python objects and modules, closes files and sockets through MicroPython cleanup, releases camera, display, preview, USB, PWM, timer, UART, thread, and other managed resources, and then repeats the Python startup sequence.

Some system state can survive a soft reset, including the RTC, CPU clock configuration, and an active network interface at the IP layer. Application code must not assume that Python objects representing those resources survive; recreate the objects and verify their state after every reset.

4.8.2 ESP-VISION Startup Order

After a hard or soft reset, ESP-VISION executes the following startup flow:


Hard or soft reset
Initialize MicroPython r untime ...
Initialize camera, displ ay, ...
Run frozen _boot.py mount internal Flas ...
Run frozen py_inisetu p.py ...
Mount SD card at /sdc ard ...
Run /boot.py when present
Initialize USB device
REPL mode
friendly
Run /main.py when present
exit, i ...
Enter friendly or raw REPL

The raw REPL used by host automation can skip `main.py` during a soft reset. This allows development tools to gain control without automatically starting the product application.

4.8.3 First Boot and Flash Filesystem

The frozen `_boot.py` only attempts to mount the internal Flash filesystem. ESP-VISION-specific setup is handled by `py_inisetup.py`. If the Flash boot sector is empty, it formats the configured `vfs` or `ffat` partition and mounts it at `/`. It then creates `/boot.py`, `/main.py`, `/README.txt`, and the `/.esp_vision_disk` marker when they do not already exist.

If mounting or writing the default files fails, `py_inisetup.py` attempts to repair the filesystem by formatting it and recreating the defaults. Formatting erases files stored in that filesystem, so product data that must survive filesystem recovery should be stored in a separate partition, on an SD card, or outside the device.

Existing startup files are not overwritten during a normal firmware update or soft reset. Board packages can provide board-specific default `main.py` and `README.txt` content through `boards/<BOARD>/board_inisetup.py`.

4.8.4 Using boot.py

Use `boot.py` for short, deterministic initialization that must complete before the application starts, such as selecting a product mode, preparing configuration, or bringing up a required network interface:

```
import network

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
```

`boot.py` must return and must not contain the application's permanent loop. ESP-VISION initializes the MicroPython USB device only after `boot.py` completes, so a blocked or long-running `boot.py` can prevent the USB REPL and host tools from becoming available.

4.8.5 Using main.py

Use `main.py` as the product application entry point. Keep the implementation in a separate module so startup policy and application logic remain independent:

```
import sys
import my_app

try:
    my_app.main()
except KeyboardInterrupt:
    raise
except Exception as error:
    print("Fatal application error:")
    sys.print_exception(error)
```

Allowing `KeyboardInterrupt` to propagate lets `Ctrl-C` stop the application and enter the friendly REPL. A production application can instead log the exception and call `machine.reset()` when automatic recovery is required, but an unconditional reset loop can make development and failure diagnosis difficult.

The default ESP-VISION `main.py` prints a board-ready message and sleeps in a loop so the VSCode extension and other host tools can take control. Press `Ctrl-C` to interrupt it and reach the REPL, or replace `/main.py` with the product entry point.

4.8.6 REPL and Recovery

`main.py` exiting normally or raising an uncaught exception transfers control to the friendly REPL. `Ctrl-C` injects `KeyboardInterrupt` into a running Python script, while `Ctrl-D` at the friendly REPL starts a soft reset.

If an application prevents normal startup, connect to the REPL, interrupt it with `Ctrl-C`, and rename or remove the startup file:

```
import os

print(os.listdir("/"))
os.rename("/main.py", "/main.disabled.py")
# Use Ctrl-D to start again without running the previous main.py.
```

If the REPL cannot be recovered, erase and reflash the board from the repository root. `erase-flash` removes the entire device Flash, including the internal filesystem and user data:

```
idf.py --board <BOARD> -p <PORT> erase-flash
idf.py --board <BOARD> -p <PORT> flash monitor
```

Reflashing without `erase-flash` normally preserves the data filesystem and therefore may preserve a faulty `boot.py` or `main.py`.

Chapter 5

API Reference

ESP-VISION provides Python APIs for camera capture, image processing, display, video encoding and streaming, and AI inference. Developers can also directly use the standard MicroPython APIs enabled in the firmware for hardware control, networking, file access, system services, and general application development. ESP-VISION maintains compatibility with commonly used OpenMV module names where applicable, including `sensor` and `image`, and additionally provides modules such as `display` and `espdl`.

The ESP-VISION module pages are kept in sync with the type stubs under `stubs/`, which are also usable for IDE completion. Their availability is derived from `micropython.cmake` during the documentation build. The *MicroPython APIs* page indexes the language, standard-library, hardware, networking, filesystem, and runtime APIs that ESP-VISION firmware inherits from the pinned MicroPython version.

5.1 MicroPython APIs

ESP-VISION is built on the MicroPython ESP32 port and directly retains its language, standard-library, hardware-control, networking, filesystem, and runtime APIs. These APIs can be used together with `sensor`, `image`, `display`, `espdl`, and the other ESP-VISION modules in the same script; ESP-VISION does not wrap or rename them.

5.1.1 Documentation Baseline

ESP-VISION pins MicroPython v1.28.0. Use the MicroPython v1.28.0 documentation as the authoritative API reference, especially the MicroPython libraries and ESP32 quick reference. MicroPython implements a resource-oriented subset of CPython, so CPython documentation must not be treated as an exact description of module contents or behavior.

5.1.2 Language and Built-in APIs

ESP-VISION retains the MicroPython language and built-in types, functions, exceptions, iteration protocol, context managers, classes, and asynchronous syntax. See MicroPython language and implementation and builtins for details.

Use exceptions and context managers to keep long-running vision applications recoverable and to close files deterministically:

```
try:
    with open("/sdcard/result.txt", "w") as output:
        output.write("capture complete\n")
```

(continues on next page)

(continued from previous page)

```
except OSError as error:
    print("storage error:", error)
```

5.1.3 Standard and Micro Libraries

Common inherited modules include `array`, `asyncio`, `binascii`, `cmath`, `collections`, `errno`, `gc`, `gzip`, `hashlib`, `heapq`, `io`, `json`, `marshal`, `math`, `os`, `platform`, `random`, `re`, `select`, `socket`, `struct`, `sys`, `time`, `weakref`, `zlib`, and `_thread`. ESP-VISION board manifests also freeze MicroPython's `asyncio` package into the firmware.

Refer to the MicroPython library index for each module's classes and functions. A documented upstream module may still be omitted or reduced by the selected firmware configuration.

5.1.4 JSON and Time Example

```
import json, time

result = {
    "timestamp_ms": time.ticks_ms(),
    "objects": 3,
    "confidence": 0.91,
}
payload = json.dumps(result)
print(payload)
```

`time.ticks_ms()` is suitable for measuring intervals and generating local runtime timestamps; use `time.ticks_diff()` when subtracting tick values so wraparound is handled correctly. `json` is useful for serializing inference results before storage or network transmission.

5.1.5 Hardware and ESP32 APIs

The inherited machine module provides MCU control and peripheral classes including `Pin`, `Signal`, `ADC`, `ADCBlock`, `PWM`, `I2C`, `SoftI2C`, `SPI`, `SoftSPI`, `UART`, `Timer`, `RTC`, `WDT`, `SDCard`, and, where supported, `DAC`, `I2S`, and `I2CTarget`. Pin assignments and peripheral conflicts remain board-specific, so consult the board schematic before combining these APIs with camera, display, storage, or codec hardware.

The inherited esp and esp32 modules expose ESP32-port-specific system and peripheral functions. Availability of individual classes such as `esp32.PCNT`, `esp32.RMT`, and other SoC-specific blocks depends on the selected chip and ESP-IDF version.

5.1.6 GPIO and PWM Example

```
from machine import Pin, PWM

led = Pin(LED_GPIO, Pin.OUT)
led.value(1)

pwm = PWM(Pin(PWM_GPIO), freq=1000, duty_u16=32768)
pwm.duty_u16(49152)
pwm.deinit()
```

Replace `LED_GPIO` and `PWM_GPIO` with pins that are free on the selected board. Check the schematic and board configuration first because camera, display, storage, USB, and codec peripherals may already reserve GPIOs.

5.1.7 Networking APIs

The inherited network, socket, and select modules provide network-interface management and TCP/UDP communication. `network.WLAN` is enabled in the current product board configurations, but successful use still requires compatible networking hardware and firmware configuration.

The current board profiles disable bluetooth and espnow. SSL/TLS, `cryptolib`, WebSocket, WebREPL, and socket-event support depend on the ESP-IDF version used to build the firmware; see *Chip and Board Support* for the maintained support summary.

5.1.8 Connect to Wi-Fi

```
import network, time

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect("ssid", "password")

for _ in range(100):
    if wlan.isconnected():
        break
    time.sleep_ms(100)

if not wlan.isconnected():
    raise OSError("Wi-Fi connection failed")
print("address:", wlan.ifconfig()[0])
```

Wi-Fi requires compatible hardware and board configuration. Production code should apply a timeout, avoid printing credentials, and handle reconnects rather than waiting indefinitely.

5.1.9 Filesystem and Runtime APIs

ESP-VISION retains MicroPython virtual-filesystem and stream behavior, including os, io, vfs, and the built-in `open()` function. Board startup code mounts product storage such as `/flash` and `/sdcard` when the corresponding partition and hardware are available.

Runtime services remain available through micropython, gc, `sys`, `time`, and `_thread`. ESP-VISION enables MicroPython threading with a GIL; vision objects and hardware resources should still be treated as shared resources unless an API explicitly documents concurrent use.

5.1.10 List Storage and Check Memory

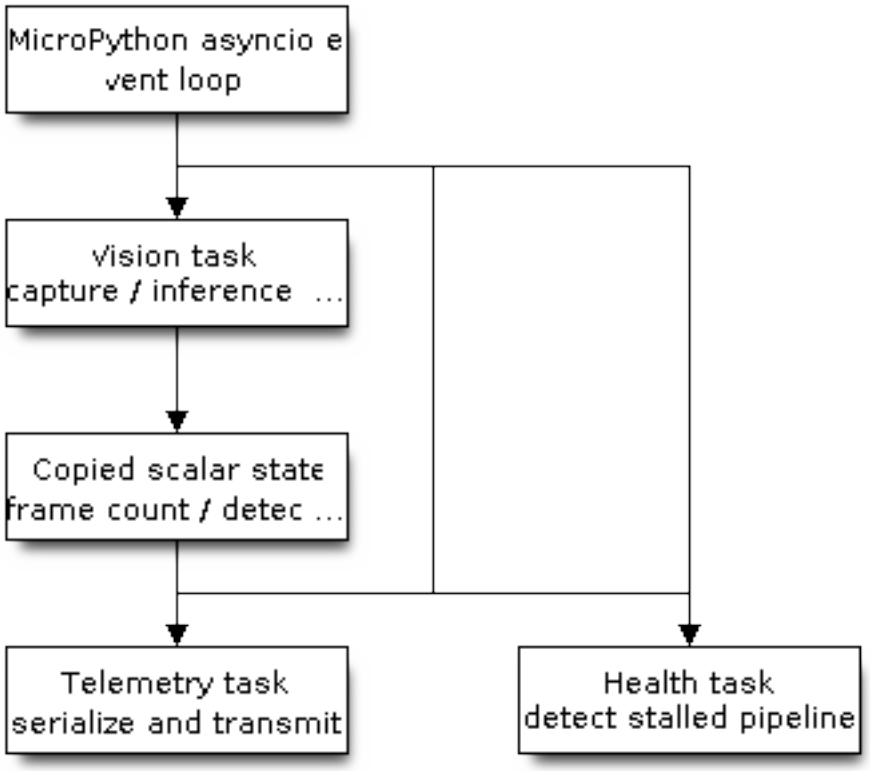
```
import gc, os

print("flash:", os.listdir("/flash"))
if "sdcard" in os.listdir("/"):
    print("sdcard:", os.listdir("/sdcard"))

gc.collect()
print("free heap:", gc.mem_free())
```

Check for optional mount points before using them. `gc.collect()` can be useful before loading a large model or allocating a frame history, but repeatedly forcing collection in every frame loop may reduce throughput.

5.1.11 Structuring a Complex Application

For developers familiar with ESP-IDF, an `asyncio` task is a cooperatively scheduled Python coroutine, not a FreeRTOS task. Multiple coroutines share one Python execution context and switch only when they `await`. This is the preferred structure for control logic, network I/O, status reporting, and time-based work because it avoids concurrent access to camera, model, display, and frame-buffer objects.

The following structure assigns exclusive ownership of the vision pipeline to one coroutine while telemetry and health-monitoring coroutines consume only copied scalar results:




```
import asyncio
import espdl
import json
import sensor
import time

MODEL = "/sdcard/espdet_pico_224_224_face.espdl"

state = {
    "frames": 0,
    "detections": 0,
    "last_frame_ms": time.ticks_ms(),
}
state_lock = asyncio.Lock()

async def vision_task(detector):
    while True:
        img = sensor.snapshot()
        results = detector.detect(img)
```

(continues on next page)

(continued from previous page)

```
        for x, y, w, h, score, category in results:
            img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
        img.flush()

        async with state_lock:
            state["frames"] += 1
            state["detections"] = len(results)
            state["last_frame_ms"] = time.ticks_ms()

        # Let ready control and network tasks run between frames.
        await asyncio.sleep_ms(0)

async def telemetry_task():
    while True:
        await asyncio.sleep_ms(1000)
        async with state_lock:
            payload = json.dumps({
                "frames": state["frames"],
                "detections": state["detections"],
            })
        print("telemetry:", payload)
        # Replace print() with non-blocking socket transmission.

async def health_task():
    while True:
        await asyncio.sleep_ms(500)
        async with state_lock:
            age = time.ticks_diff(
                time.ticks_ms(), state["last_frame_ms"]
            )
        if age > 3000:
            print("warning: vision pipeline stalled")

async def main():
    sensor.reset()
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QVGA)
    sensor.skip_frames(time=1000)
    detector = espdl.ESPDet(MODEL, score=0.5, nms=0.7)

    vision = asyncio.create_task(vision_task(detector))
    telemetry = asyncio.create_task(telemetry_task())
    health = asyncio.create_task(health_task())
    try:
        await asyncio.gather(vision, telemetry, health)
    finally:
        vision.cancel()
        telemetry.cancel()
        health.cancel()
        detector.deinit()

asyncio.run(main())
```

Calls such as `sensor.snapshot()`, inference, image processing, and `img.flush()` are synchronous. While one of these calls is running, other coroutines cannot run; the explicit `await asyncio.sleep_ms(0)` provides a scheduling point between frames. If a network operation can block, use an `asyncio` stream or non-blocking socket rather than a long blocking call.

5.1.12 Threads and Native Tasks

The inherited `_thread.start_new_thread()` API creates another FreeRTOS task, but it is not equivalent to `xTaskCreatePinnedToCore()` as an application interface. Python does not expose task priority or core affinity here; MicroPython threads are pinned to `MP_TASK_COREID` and share the interpreter through the GIL. `_thread.stack_size()` can configure the stack allocated for subsequently created threads.

Use `_thread` only when a blocking operation cannot be integrated with `asyncio`, protect shared Python data with `_thread.allocate_lock()`, and keep camera, display, codec, model, and frame-buffer ownership in one thread unless an API explicitly documents thread safety. An IRQ or another thread can notify the event loop with `asyncio.ThreadSafeFlag` without directly manipulating vision objects.

For deterministic latency, explicit FreeRTOS priorities or core affinity, ISR-to-task notification, or continuous work that must proceed while Python holds the GIL, implement the worker as an ESP-IDF C/C++ component and expose a narrow Python API. In ESP-IDF terms, use `asyncio` tasks for application state machines, `_thread` for exceptional blocking integration, and native tasks for real-time services.

5.1.13 Check Availability on a Firmware

Standard MicroPython API availability is controlled by `overlay/micropython/ports/esp32/mpconfigport.h`, the selected board's `mpconfigboard.h` and `mpconfigboard.cmake`, ESP-IDF version checks, and SoC capability macros. Because availability is not determined solely by the chip, the definitive check is the exact firmware:

```
help("modules")
import machine
help(machine)
```

Use `hasattr()` or a guarded import when an application must run across multiple ESP-VISION boards. To remove or add inherited APIs in a product firmware, follow *Customize Firmware Features*.

See also:

Runnable examples that use only standard MicroPython APIs: `example/00-HelloWorld/helloworld.py` (first script), `example/06-Peripherals/00-Storage/sdcard.py` (filesystem), and `example/06-Peripherals/02-WiFi/webrepl.py` (Wi-Fi and WebREPL).

5.2 sensor –Camera

The `sensor` module controls the camera and captures frames. It mirrors the OpenMV `sensor` API so existing OpenMV scripts port with little change.

5.2.1 Camera Initialization and Continuous Capture

A typical program resets the sensor, selects a pixel format and frame size, lets automatic exposure and white balance settle, and then captures frames continuously:

```
import sensor

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=2000)

while True:
    img = sensor.snapshot()
```

`sensor.reset()` applies the board's default camera configuration. `RGB565` is suitable for display, drawing, and most AI workflows, while `GRAYSCALE` reduces memory and processing cost for many detection algorithms. `sensor.snapshot()` returns an *`image.Image`* backed by the reusable frame buffer, so copy the image when it must remain unchanged after the next capture.

5.2.2 Image Orientation and Camera Status

Use horizontal mirror and vertical flip to correct the image orientation imposed by the physical sensor installation. `status()` provides the active dimensions, pixel format, sensor ID, orientation, and crop information for diagnostics:

```
import sensor

sensor.reset()
sensor.set_hmirror(True)
sensor.set_vflip(False)

info = sensor.status()
print("sensor id:", info["id"])
print("output:", info["width"], "x", info["height"])
print("ready:", info["ready"])
```

Mirror and flip settings affect subsequently captured frames. Product code can call `status()` after initialization to verify that the camera is ready and that the negotiated output size matches the processing pipeline.

5.2.3 Temporarily Stop Capture

The camera stream can be stopped when capture is not required and restarted before the next frame:

```
sensor.shutdown(True)
# Perform work that does not require the camera.
sensor.shutdown(False)
sensor.skip_frames(n=3)
img = sensor.snapshot()
```

After restarting the stream, discard a few frames if exposure or the incoming frame queue needs to stabilize.

See also:

The Camera Pipeline explains how a frame travels from the image sensor to an *`image.Image`*, and *The Image Model* covers pixel formats and color spaces.

Runnable examples: `example/01-Camera/00-Snapshot` (capture and save) and `example/01-Camera/03-MJPEG` (Wi-Fi MJPEG stream).

5.2.4 Constants

`sensor.`**`GRAYSCALE`**

Pixel format constants accepted by set_pixformat().

`sensor.`**`RGB565`**

`sensor.`**`QQVGA`**

Frame size constants accepted by set_framesize().

`sensor.`**`QVGA`**

5.2.5 Functions

`sensor.`**`reset`**`()`

Reset and initialize the camera with the board default sensor configuration.

`sensor.`**`shutdown`**`(`*enable=...*`)`

Stop or restart the camera stream.

Parameters

`enable` –True shuts the camera down; False starts it again.

`sensor.`**`set_pixformat`**`(`*pixformat*`)`

Select the output pixel format.

Parameters

`pixformat` –sensor.GRAYSCALE or sensor.RGB565.

`sensor.`**`get_pixformat`**`()`

Return the current pixel format constant.

`sensor.`**`set_framesize`**`(`*framesize*`)`

Select the output frame size.

Parameters

`framesize` –sensor.QQVGA or sensor.QVGA.

`sensor.`**`get_framesize`**`()`

Return the current frame size constant.

`sensor.`**`width`**`()`

Return the current output image width in pixels.

`sensor.`**`height`**`()`

Return the current output image height in pixels.

`sensor.`**`get_id`**`()`

Return the camera sensor ID.

`sensor.`**`set_hmirror`**`(`*enable*`)`

Mirror the camera image horizontally.

Parameters

`enable` –True enables horizontal mirror.

`sensor.`**`get_hmirror`**`()`

Return whether horizontal mirror is enabled.

`sensor.`**`set_vflip`**`(`*enable*`)`

Flip the camera image vertically.

Parameters

`enable` –True enables vertical flip.

`sensor.`**`get_vflip`**`()`

Return whether vertical flip is enabled.

`sensor.`**`skip_frames`**`(`*time=..., n=...*`)`

Drop frames while camera exposure and processing settle.

Parameters

`time` –milliseconds to wait. n: number of frames to skip.

`sensor.`**`snapshot`**`(`*buffer*`=...)`

Capture one image from the camera.

Parameters

`buffer` –reserved for compatibility; pass None in current ESP-VISION builds.

`sensor.`**`status`**`()`

Return camera readiness, size, format, mirror, flip, and crop status.

5.2.6 Classes

`class` `sensor.`**`SensorStatus`**

Dictionary returned by status().

5.3 image –Image Processing

The `image` module provides the `Image` object and the vision algorithms built on OpenMV `imlib`: drawing, format conversion, filtering, color/blob analysis, and feature detection (lines, circles, rectangles, QR codes, and AprilTags). The codec stream type *`imageio.ImageIO`* is documented in *image.ImageIO –Image Stream*.

The current ESP32-P4 board profiles also enable the ZXing-C++ barcode backend and *`image.Image.find_barcodes()`*.

5.3.1 Drawing and Color-Blob Tracking

```
import sensor

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

img = sensor.snapshot()
img.draw_rectangle(10, 10, 50, 50, color=(255, 0, 0))
thresholds = [(30, 100, 15, 127, 15, 127)]
for blob in img.find_blobs(thresholds, pixels_threshold=80, area_threshold=80):
    img.draw_rectangle(blob.rect(), color=(255, 0, 0), thickness=2)
    img.draw_cross(blob.cx(), blob.cy(), color=(0, 255, 0))
```

An RGB565 threshold contains six LAB limits: `(L_min, L_max, A_min, A_max, B_min, B_max)`. `pixels_threshold` rejects regions containing too few matching pixels, while `area_threshold` rejects regions whose bounding box is too small. Blob coordinates are returned in source-image pixels and can be passed directly to drawing methods.

5.3.2 Filtering and Binary Segmentation

```
sensor.set_pixformat(sensor.GRAYSCALE)
img = sensor.snapshot()
img.gaussian(1)
img.binary([(80, 255)])
img.erode(1)
img.dilate(1)
img.flush()
```

`gaussian()` suppresses high-frequency noise before thresholding. `binary()` converts matching grayscale values to the active binary value, and the erosion/dilation pair removes isolated pixels and restores the main regions. These methods modify the image in place.

5.3.3 QR Code Recognition

```
sensor.set_pixformat(sensor.GRAYSCALE)
img = sensor.snapshot()
for code in img.find_qrcodes():
    img.draw_rectangle(code.rect(), color=255, thickness=2)
    print(code.payload())
```

`find_qrcodes()` returns geometry and decoded payload data. Grayscale input generally reduces processing cost, and a smaller ROI can be supplied when the code is expected in a known part of the frame.

5.3.4 Line and Circle Detection

```
img = sensor.snapshot()
for line in img.find_lines(threshold=1400, theta_margin=25, rho_margin=25):
    img.draw_line(line.line(), color=255, thickness=2)

for circle in img.find_circles(threshold=2500, r_min=8, r_max=80, r_step=4):
    img.draw_circle(circle.x(), circle.y(), circle.r(), color=255, thickness=2)
```

Detection thresholds control the required accumulator strength. Increasing them reduces weak detections; constraining the radius range, ROI, or image resolution lowers computation and usually improves stability.

5.3.5 Encode and Save an Image

```
import image

img = sensor.snapshot()
jpg = img.to_jpeg(quality=85, subsampling=image.JPEG_SUBSAMPLING_420)
jpg.save("/sdcard/snapshot.jpg")
print("encoded bytes:", jpg.size())
```

JPEG quality trades encoded size for detail, and 4:2:0 chroma subsampling usually produces the smallest color image. Saving requires the destination filesystem to be mounted and writable.

Most in-place methods return the image itself, so operations can be chained: `img.to_grayscale().gaussian(1).binary([(128, 255)])`.

See also:

For the theory behind these methods, see *The Image Model* (pixel formats, color spaces, the frame buffer) and *Image Processing* (filtering, thresholding, feature detection).

Runnable examples: `example/02-Image-Processing` (drawing, filters, color tracking, frame differencing), `example/04-Barcodes` (QR codes), and `example/05-Feature-Detection` (AprilTags, lines, circles).

5.3.6 Constants

`image.`**`BINARY`**

1-bpp black-and-white pixel format (one bit per pixel).

`image.`**`GRAYSCALE`**
: 8-bit grayscale pixel format.

`image.`**`RGB565`**
: 16-bit RGB565 color pixel format.

`image.`**`BAYER`**
: Raw Bayer mosaic pixel format from the sensor.

`image.`**`YUV422`**
: Packed YUV422 pixel format.

`image.`**`JPEG`**
: JPEG-compressed image format.

`image.`**`PNG`**
: PNG-compressed image format.

`image.`**`PALETTE_RAINBOW`**
: False-color rainbow palette (cold to hot).

`image.`**`PALETTE_IRONBOW`**
: "Ironbow" thermal palette.

`image.`**`PALETTE_DEPTH`**
: Depth-map palette.

`image.`**`PALETTE_EVT_DARK`**
: Event-camera dark palette.

`image.`**`PALETTE_EVT_LIGHT`**
: Event-camera light palette.

`image.`**`AREA`**
: Scaling/geometry hint flags for copy()/crop()/scale()/draw_image(): area-average downscaling.

`image.`**`BILINEAR`**
: Bilinear interpolation hint.

`image.`**`BICUBIC`**
: Bicubic interpolation hint.

`image.`**`HMIRROR`**
: Mirror horizontally.

`image.`**`VFLIP`**
: Flip vertically.

`image.`**`TRANSPOSE`**
: Swap X and Y (transpose).

`image.`**`CENTER`**
: Center the source in the destination.

`image.`**`EXTRACT_RGB_CHANNEL_FIRST`**
: Extract the RGB channel before applying the color palette.

`image.`**`APPLY_COLOR_PALETTE_FIRST`**
: Apply the color palette before other steps.

`image.`**`SCALE_ASPECT_KEEP`**
: Keep aspect ratio, fitting inside the destination.

`image.`**`SCALE_ASPECT_EXPAND`**
: Keep aspect ratio, expanding to fill the destination.

`image.`**`SCALE_ASPECT_IGNORE`**
: Ignore aspect ratio, stretching to the destination.

`image.`**`BLACK_BACKGROUND`**
: Treat the background as black (needed for correct alpha blending).

`image.`**`ROTATE_90`**
: Rotate 90 degrees.

`image.`**`ROTATE_180`**
: Rotate 180 degrees.

`image.`**`ROTATE_270`**
: Rotate 270 degrees.

`image.`**`JPEG_SUBSAMPLING_AUTO`**
: Let the JPEG encoder pick chroma subsampling automatically.

`image.`**`JPEG_SUBSAMPLING_444`**
: 4:4:4 JPEG chroma subsampling (best quality).

`image.`**`JPEG_SUBSAMPLING_422`**
: 4:2:2 JPEG chroma subsampling.

`image.`**`JPEG_SUBSAMPLING_420`**
: 4:2:0 JPEG chroma subsampling (smallest size).

`image.`**`SEARCH_EX`**
: Exhaustive template-search strategy.

`image.`**`SEARCH_DS`**
: Diamond-search (faster, approximate) template-search strategy.

`image.`**`EDGE_CANNY`**
: Canny edge-detection method for find_edges().

`image.`**`EDGE_SIMPLE`**
: Simple gradient-threshold edge-detection method for find_edges().

`image.`**`CORNER_FAST`**
: FAST corner detector for find_keypoints().

`image.`**`CORNER_AGAST`**
: AGAST corner detector for find_keypoints().

`image.`**`EAN2`**
: Barcode symbology constants returned by barcode.type() / accepted by find_barcodes().

`image.`**`EAN5`**

`image.`**`EAN8`**

`image.`**`UPCE`**

`image.`**`ISBN10`**

`image.`**`UPCA`**

`image.`**`EAN13`**

image.**ISBN13**

image.**I25**

image.**DATABAR**

image.**DATABAR_EXP**

image.**CODABAR**

image.**CODE39**

image.**PDF417**

image.**CODE93**

image.**CODE128**

image.**TAG16H5**

AprilTag family constants accepted by find_apriltags().

image.**TAG25H7**

image.**TAG25H9**

image.**TAG36H10**

image.**TAG36H11**

5.3.7 Functions

image.**binary_to_grayscale**(*pixel*)

Convert a 1-bpp binary pixel to a grayscale value.

image.**binary_to_rgb**(*pixel*)

Convert a 1-bpp binary pixel to an (r, g, b) tuple.

image.**binary_to_lab**(*pixel*)

Convert a 1-bpp binary pixel to a LAB tuple.

image.**binary_to_yuv**(*pixel*)

Convert a 1-bpp binary pixel to a YUV tuple.

image.**grayscale_to_binary**(*pixel*)

Convert a grayscale value to a 1-bpp binary pixel.

image.**grayscale_to_rgb**(*pixel*)

Convert a grayscale value to an (r, g, b) tuple.

image.**grayscale_to_lab**(*pixel*)

Convert a grayscale value to a LAB tuple.

image.**grayscale_to_yuv**(*pixel*)

Convert a grayscale value to a YUV tuple.

image.**rgb_to_binary**(*pixel*)

Convert an (r, g, b) color to a 1-bpp binary pixel.

image.**rgb_to_grayscale**(*pixel*)

Convert an (r, g, b) color to a grayscale value.

`image.`**`rgb_to_lab`**(*pixel*)
Convert an (r, g, b) color to a LAB tuple.

`image.`**`rgb_to_yuv`**(*pixel*)
Convert an (r, g, b) color to a YUV tuple.

`image.`**`lab_to_binary`**(*pixel*)
Convert a LAB tuple to a 1-bpp binary pixel.

`image.`**`lab_to_grayscale`**(*pixel*)
Convert a LAB tuple to a grayscale value.

`image.`**`lab_to_rgb`**(*pixel*)
Convert a LAB tuple to an (r, g, b) tuple.

`image.`**`lab_to_yuv`**(*pixel*)
Convert a LAB tuple to a YUV tuple.

`image.`**`yuv_to_binary`**(*pixel*)
Convert a YUV tuple to a 1-bpp binary pixel.

`image.`**`yuv_to_grayscale`**(*pixel*)
Convert a YUV tuple to a grayscale value.

`image.`**`yuv_to_rgb`**(*pixel*)
Convert a YUV tuple to an (r, g, b) tuple.

`image.`**`yuv_to_lab`**(*pixel*)
Convert a YUV tuple to a LAB tuple.

5.3.8 Classes

`class` `image.`**`line`**
A detected line segment with Hough-space attributes.

`line`()
Return the line as (x1, y1, x2, y2).

`x1`()
Start-point x coordinate.

`y1`()
Start-point y coordinate.

`x2`()
End-point x coordinate.

`y2`()
End-point y coordinate.

`length`()
Segment length in pixels.

`magnitude`()
Hough accumulator magnitude (line strength).

`theta`()
Line angle theta in degrees (Hough space).

`rho`()
Line distance rho in pixels (Hough space).

class image.**circle**

A detected circle.

circle()

Return the circle as (x, y, r).

x()

Center x coordinate.

y()

Center y coordinate.

r()

Radius in pixels.

magnitude()

Hough accumulator magnitude (circle strength).

class image.**rect**

A detected rectangle (quadrilateral).

corners()

Return the four corner points in clockwise order.

rect()

Return the bounding box as (x, y, w, h).

x()

Bounding-box top-left x.

y()

Bounding-box top-left y.

w()

Bounding-box width.

h()

Bounding-box height.

magnitude()

Detection strength.

class image.**blob**

A connected color region found by find_blobs().

corners()

The four bounding-box corners.

min_corners()

The four corners of the minimum-area enclosing rectangle.

rect()

Bounding box as (x, y, w, h).

x()

Bounding-box top-left x.

y()

Bounding-box top-left y.

w()

Bounding-box width.

h()
Bounding-box height.

pixels()
Number of pixels in the blob.

cx()
Centroid x (integer).

cxf()
Centroid x (float).

cy()
Centroid y (integer).

cyf()
Centroid y (float).

rotation()
Orientation of the blob’ s major axis in radians.

rotation_deg()
Orientation in degrees.

rotation_rad()
Orientation in radians.

code()
Bitmask of the thresholds this blob matched.

count()
Number of blobs merged into this one.

perimeter()
Bounding-box perimeter.

roundness()
Roundness metric in [0, 1] (1 == circle).

elongation()
Elongation metric in [0, 1].

area()
Bounding-box area (w * h).

density()
Pixel density: pixels / area.

extent()
Alias of density().

compactness()
Compactness metric in [0, 1].

solidity()
Solidity (pixels / convex-hull area) in [0, 1].

convexity()
Convexity metric in [0, 1].

x_hist_bins()
Horizontal projection histogram bins.

y_hist_bins()
Vertical projection histogram bins.

major_axis_line()
Major-axis line as (x1, y1, x2, y2).

minor_axis_line()
Minor-axis line as (x1, y1, x2, y2).

enclosing_circle()
Minimum enclosing circle as (x, y, r).

enclosed_ellipse()
Best-fit enclosed ellipse as (x, y, rx, ry, rotation).

class image.**qrcode**
A decoded QR code found by find_qrcodes().

corners()
The four corner points.

rect()
Bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

payload()
Decoded payload string.

version()
QR version (size) number.

ecc_level()
Error-correction level.

mask()
Data mask pattern index.

data_type()
Encoding mode of the payload.

eci()
Extended Channel Interpretation value.

is_numeric()
True if the payload is numeric.

is_alphanumeric()
True if the payload is alphanumeric.

is_binary()
True if the payload is binary.

is_kanji()
True if the payload is Kanji.

class image.**barcode**
A decoded 1D/2D barcode found by find_barcodes().

corners()
The four corner points.

rect()
Bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

payload()
Decoded payload string.

type()
Symbology type constant (e.g. EAN13, CODE128).

rotation()
Rotation of the barcode in degrees.

quality()
Decode quality (number of agreeing scan lines).

class image.**apriltag**
A detected AprilTag with optional 6-DoF pose. Attributes are fields, not methods.

class image.**histogram**
Channel histogram returned by get_histogram(). For RGB565 the L/A/B channels correspond to the LAB color space.

bins()
Combined bins (grayscale/binary) or L bins (RGB565).

l_bins()
L-channel (lightness) bins.

a_bins()
A-channel bins (RGB565 only).

b_bins()
B-channel bins (RGB565 only).

get_percentile(*p*)
Return the value at the given cumulative percentile (0..1).

Parameters
p –percentile in the range 0 to 1.

get_threshold()
Compute an Otsu threshold from the histogram.

get_stats()

Compute statistics (mean/median/mode/stdev/…) from the histogram.

get_statistics()

Alias of get_stats().

statistics()

Alias of get_stats().

class image.**percentile**

A percentile result from histogram.get_percentile().

value()

Combined/L value at the percentile.

l_value()

L-channel value.

a_value()

A-channel value.

b_value()

B-channel value.

class image.**threshold**

An Otsu threshold result from histogram.get_threshold().

value()

Combined/L threshold value.

l_value()

L-channel threshold.

a_value()

A-channel threshold.

b_value()

B-channel threshold.

class image.**statistics**

Region statistics returned by get_statistics(). Combined accessors apply to grayscale/binary; `l_*`/`a_*`/`b_*` apply to RGB565 LAB channels.

mean()

median()

mode()

stdev()

min()

max()

lq()

Lower quartile (25th percentile).

uq()

Upper quartile (75th percentile).

l_mean()

l_median()

l_mode()

l_stdev()

l_min()

l_max()

l_lq()

l_uq()

a_mean()

a_median()

a_mode()

a_stdev()

a_min()

a_max()

a_lq()

a_uq()

b_mean()

b_median()

b_mode()

b_stdev()

b_min()

b_max()

b_lq()

b_uq()

class image.**Image**(*path*, *, *copy_to_fb=...*)

class image.**Image**(*width*, *height*, *pixformat*, *, *buffer=...*, *copy_to_fb=...*)

class image.**Image**(*array*, *, *buffer=...*, *copy_to_fb=...*)

A 2D image backed by a pixel buffer. Create one from a file path, from a width/height/pixformat, or from an array; most often you get one from sensor.snapshot(). Most in-place methods return `self` so calls can be chained.

Load an image from a file (BMP/PPM/PGM/JPEG/PNG depending on build).

Parameters

- **path** –source file path.
- **copy_to_fb** –True loads the decoded image into the frame buffer.

width()

Image width in pixels.

height()

Image height in pixels.

format()

Pixel format constant (BINARY/GRAYSCALE/RGB565/…).

size()

Size of the pixel buffer in bytes.

bytearray()

Return the pixel buffer as a bytearray (no copy).

get_pixel(*x*, *y*, *, *rgbtuple=...*)

Read one pixel. x, y: pixel coordinates.

Parameters

rgbtuple –for RGB565, True returns an (r, g, b) tuple, False the packed value.

set_pixel(*x*, *y*, *pixel*)

Write one pixel. x, y: pixel coordinates.

Parameters

pixel –grayscale value or (r, g, b) color.

flush()

Push this image to the host preview over USB CDC (EVFRAME JPEG stream).

save(*path*, *, *roi=...*, *quality=...*)

Save the image to a file; the format is inferred from the extension.

Parameters

- **path** –destination path.
- **roi** –optional source rectangle (x, y, w, h).
- **quality** –JPEG quality from 1 to 100 when saving as JPEG.

copy(*, *roi=...*, *x_scale=...*, *y_scale=...*, *rgb_channel=...*, *alpha=...*, *color_palette=...*, *alpha_palette=...*, *hint=...*, *transform=...*, *copy=...*)

Return a scaled/cropped/converted copy of the image. x_scale, y_scale: horizontal/vertical scale factors.

Parameters

- **roi** –source rectangle (x, y, w, h) to read from.
- **hint** –bitwise OR of AREA/BILINEAR/BICUBIC/HMIRROR/VFLIP/TRANSPOSE/ROTATE_*/SCALE_ flags.
- **rgb_channel** –extract a single RGB channel (0, 1, or 2; -1 for all).
- **alpha** –blend alpha from 0 to 255.
- **copy** –True returns a new image; False converts in place.

crop(*, *roi=...*, *x_scale=...*, *y_scale=...*, *hint=...*, *copy=...*)

Crop/scale the image to a region of interest. Same options as copy().

scale(*, *x_scale=...*, *y_scale=...*, *roi=...*, *hint=...*, *copy=...*)

Scale the image by x_scale/y_scale (alias of crop in this build).

compress(*, *quality=...*, *subsampling=...*, *roi=...*)

Compress the image to JPEG in place (alias of to_jpeg).

Parameters

- **quality** –JPEG quality from 1 to 100.
- **subsampling** –one of the JPEG_SUBSAMPLING_* constants.

to_bitmap(*, *copy=...*, *roi=...*)

Convert to a 1-bpp bitmap.

to_grayscale(*, *copy=...*, *roi=...*)

Convert to grayscale.

to_rgb565(*, *copy*=..., *roi*=...)

Convert to RGB565.

to_rainbow(*, *copy*=..., *roi*=...)

Apply the rainbow palette and convert to RGB565.

to_ironbow(*, *copy*=..., *roi*=...)

Apply the ironbow thermal palette and convert to RGB565.

to_jpeg(*, *quality*=..., *subsampling*=..., *copy*=...)

Encode to JPEG.

Parameters

- **quality** –JPEG quality from 1 to 100.
- **subsampling** –one of the JPEG_SUBSAMPLING_* constants.

to_png(*, *copy*=...)

Encode to PNG.

clear(*, *mask*=...)

Fill the image with zeros (optionally only where mask is set).

Parameters

mask –optional 1-bpp image limiting which pixels are cleared.

draw_line(*x0*, *y0*, *x1*, *y1*, *color*=..., *thickness*=...)

draw_line(*line*, *color*=..., *thickness*=...)

draw_rectangle(*x*, *y*, *w*, *h*, *color*=..., *thickness*=..., *fill*=...)

draw_rectangle(*rect*, *color*=..., *thickness*=..., *fill*=...)

draw_circle(*x*, *y*, *r*, *color*=..., *thickness*=..., *fill*=...)

draw_circle(*circle*, *color*=..., *thickness*=..., *fill*=...)

draw_ellipse(*x*, *y*, *rx*, *ry*, *rotation*, *color*=..., *thickness*=..., *fill*=...)

draw_ellipse(*ellipse*, *color*=..., *thickness*=..., *fill*=...)

draw_string(*x*, *y*, *text*, *color*=..., *scale*=..., *x_spacing*=..., *y_spacing*=..., *mono_space*=..., *char_rotation*=..., *char_hmirror*=..., *char_vflip*=..., *string_rotation*=..., *string_hmirror*=..., *string_vflip*=...)

Draw text using the built-in bitmap font. x, y: top-left position of the text.

Parameters

- **text** –string to draw.
- **color** –text color.
- **scale** –integer/float scale factor of the font.

draw_cross(*x*, *y*, *color*=..., *size*=..., *thickness*=...)

Draw a cross marker centered at (x, y).

draw_arrow(*x0*, *y0*, *x1*, *y1*, *color*=..., *size*=..., *thickness*=...)

draw_arrow(*line*, *color*=..., *size*=..., *thickness*=...)

draw_edges(*corners*, *color*=..., *size*=..., *thickness*=..., *fill*=...)

Draw connecting edges between a sequence of corner points.

draw_keypoints(*keypoints*, *color*=..., *size*=..., *thickness*=..., *fill*=...)

Draw a set of keypoints.

draw_image(*image*, *x=...*, *y=...*, ***, *x_scale=...*, *y_scale=...*, *roi=...*, *rgb_channel=...*, *alpha=...*, *color_palette=...*, *alpha_palette=...*, *hint=...*, *transform=...*, *mask=...*)

Alpha-blend/scale another image (or a solid color) onto this one. x, y: destination top-left position. x_scale, y_scale: scale factors; roi: source rectangle; alpha: 0..255;

Parameters

- **image** –source Image, an (r, g, b) tuple, or a packed scalar color.
- **hint** –SCALE_ASPECT_*/interpolation flags; mask: optional 1-bpp mask.

binary(*thresholds*, ***, *invert=...*, *zero=...*, *mask=...*, *to_bitmap=...*, *copy=...*)

Threshold the image into a binary mask against a list of color thresholds.

Parameters

- **thresholds** –list of grayscale (min, max) or LAB 6-tuple thresholds.
- **invert** –invert the match.
- **zero** –zero out matching pixels instead of setting them.
- **mask** –optional 1-bpp image limiting the operation.
- **to_bitmap** –output a 1-bpp bitmap.
- **copy** –return a new image instead of modifying in place.

invert()

Invert pixel values.

erode(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological erosion with a (2*ksize+1) square structuring element.

Parameters

- **ksize** –structuring-element radius.
- **threshold** –minimum number of set neighbors to keep a pixel.
- **mask** –optional 1-bpp image limiting the operation.

dilate(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological dilation. See erode() for parameters.

open(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological opening (erode then dilate).

close(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological closing (dilate then erode).

difference(*image*, *x=...*, *y=...*, ***, *roi=...*, *mask=...*)

Absolute per-pixel difference against another image/color (frame differencing). x, y: placement of the other image.

Parameters

image –other Image, (r, g, b) tuple, or scalar color.

histeq(***, *adaptive=...*, *clip_limit=...*, *mask=...*)

Histogram equalization (global). adaptive is not supported in this build.

Parameters

clip_limit –reserved; mask: optional 1-bpp image.

mean(*ksize*, ***, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Box (mean) blur over a (2*ksize+1) window. threshold/offset/invert: adaptive-threshold the result.

Parameters

- **ksize** –kernel radius.
- **mask** –optional 1-bpp image.

median(*ksize*, ***, *percentile=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Median filter (edge-preserving denoise).

Parameters

- **ksize** –kernel radius.
- **percentile** –rank in [0, 1] to select (0.5 == median).

mode(*ksize*, *, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Mode (most-common value) filter.

Parameters

ksize –kernel radius.

midpoint(*ksize*, *, *bias=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Midpoint filter ((min + max)/2 blended by bias).

Parameters

ksize –kernel radius; bias: 0 == min, 1 == max, 0.5 == midpoint.

morph(*ksize*, *kernel*, *, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Apply an arbitrary NxN convolution kernel.

Parameters

- **ksize** –kernel radius; the kernel must be (2*ksize+1) square.
- **kernel** –flat or nested sequence of integer weights.
- **mul** –output multiplier (defaults to 1/sum(kernel)); add: output bias.

blur(*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur using a separable Pascal-triangle kernel (alias: gaussian, gaussian_blur).

Parameters

- **ksize** –kernel radius.
- **unsharp** –True turns the filter into an unsharp-mask sharpener.

gaussian(*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur. See blur() for parameters.

gaussian_blur(*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur. See blur() for parameters.

laplacian(*ksize*, *, *sharpen=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Laplacian edge filter (or sharpener when sharpen is True).

Parameters

ksize –kernel radius.

bilateral(*ksize*, *, *color_sigma=...*, *space_sigma=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Bilateral filter (edge-preserving smoothing).

Parameters

- **ksize** –kernel radius.
- **color_sigma** –range sigma (color closeness); larger blurs across more contrast.
- **space_sigma** –spatial sigma (distance closeness).

get_histogram(*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Compute a channel histogram over the image or a region. bins / l_bins / a_bins / b_bins: bin counts; -1 selects the channel default.

Parameters

- **thresholds** –optional list of thresholds to restrict counted pixels.
- **invert** –invert the threshold match.
- **roi** –region of interest (x, y, w, h).
- **difference** –optional image to histogram the difference against.

get_hist(*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_histogram().

histogram(*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_histogram().

get_statistics(*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Compute region statistics (mean/median/mode/stdev/quartiles). thresholds/invert/roi/bins: as in get_histogram().

get_stats(*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_statistics().

statistics(*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_statistics().

get_regression(*thresholds*, *, *invert=...*, *roi=...*, *x_stride=...*, *y_stride=...*, *area_threshold=...*, *pixels_threshold=...*, *robust=...*)

Fit a line to the thresholded pixels via least-squares (or robust) regression. x_stride, y_stride: pixel sampling steps. area_threshold, pixels_threshold: minimum region area / pixel count to fit.

Parameters

- **thresholds** –list of color thresholds selecting the pixels to fit.
- **invert** –invert the match; roi: region of interest.
- **robust** –use a robust (Theil-Sen) estimator.

find_blobs(*thresholds*, *, *invert=...*, *roi=...*, *x_stride=...*, *y_stride=...*, *area_threshold=...*, *pixels_threshold=...*, *merge=...*, *margin=...*, *threshold_cb=...*, *merge_cb=...*, *x_hist_bins_max=...*, *y_hist_bins_max=...*)

Find connected color regions (blobs) matching the thresholds. x_stride, y_stride: pixel sampling steps. area_threshold, pixels_threshold: minimum bounding-box area / pixel count. threshold_cb / merge_cb: optional Python filter/merge callbacks.

Parameters

- **thresholds** –list of color thresholds.
- **invert** –invert the match; roi: region of interest.
- **merge** –merge overlapping blobs; margin: extra merge margin.

find_lines(*, *roi=...*, *x_stride=...*, *y_stride=...*, *threshold=...*, *theta_margin=...*, *rho_margin=...*)

Find straight lines with the Hough transform. theta_margin, rho_margin: merge tolerance for similar lines.

Parameters

- **roi** –region of interest; x_stride, y_stride: sampling steps.
- **threshold** –minimum Hough accumulator magnitude to report a line.

find_circles(*, *roi=...*, *x_stride=...*, *y_stride=...*, *threshold=...*, *x_margin=...*, *y_margin=...*, *r_margin=...*, *r_min=...*, *r_max=...*, *r_step=...*)

Find circles with the Hough transform. x_margin, y_margin, r_margin: merge tolerances. r_min, r_max, r_step: radius search range and step.

Parameters

threshold –minimum accumulator magnitude.

find_rects(*, *roi=...*, *threshold=...*)

Find rectangles/quadrilaterals (e.g. for fiducials).

Parameters

threshold –minimum edge-magnitude score.

find_qrcodes(*, *roi=...*)

Find and decode QR codes.

Parameters

roi –region of interest.

find_barcodes(*, *roi=...*)

Find and decode 1D/2D barcodes (requires the barcode backend).

Parameters

roi –region of interest.

find_apriltags(*, *roi=...*, *families=...*, *fx=...*, *fy=...*, *cx=...*, *cy=...*, *pose=...*)

Find and decode AprilTags, optionally estimating 6-DoF pose. fx, fy, cx, cy: camera intrinsics for pose estimation.

Parameters

- **roi** –region of interest; families: OR of TAG* family constants.
- **pose** –True computes translation/rotation (needs intrinsics).

5.4 display –LCD Output

The `display` module drives the board LCD. Construct one `ESP32Display` (aliased as `display.Display`) and push frames to it with `ESP32Display.write()`.

5.4.1 Camera Preview

```
import sensor, display

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)

lcd = display.ESP32Display()
while True:
    lcd.write(sensor.snapshot(), fit=True)
```

With `fit=True`, the source image is scaled to fit the panel while preserving the display driver's default placement behavior. Create the display once and reuse it; repeatedly constructing the object reinitializes panel resources.

5.4.2 Position, Scale, and Crop

Use `x` and `y` for placement, `x_scale` and `y_scale` for explicit scaling, and `roi` to display only a source region:

```
img = sensor.snapshot()
lcd.write(
    img,
    x=20,
    y=10,
    x_scale=0.5,
    y_scale=0.5,
    roi=(40, 30, 160, 120),
    fit=False,
)
```

When `fit=False`, the explicit position and scale values control the output. The ROI is expressed in source-image coordinates as `(x, y, width, height)`; reducing the ROI also reduces the amount of image data sent to the display pipeline.

5.4.3 Backlight and Cleanup

```
lcd = display.Display(backlight=80)
print("panel:", lcd.width(), "x", lcd.height())
lcd.backlight(30)
lcd.clear()
lcd.deinit()
```

Backlight values are percentages from 0 to 100. Call `deinit()` when an application permanently releases the display; do not write frames after the driver has been deinitialized.

See also:

Runnable example: `example/06-Peripherals/01-Display/lcd_preview.py`.

5.4.4 Classes

class display.**ESP32Display**(*width=...*, *height=...*, *refresh=...*, *, *backlight=...*)

ESP32 display object for the board LCD.

Create and initialize the display.

Parameters

- **width** –requested display width; 0 uses board default.
- **height** –requested display height; 0 uses board default.
- **refresh** –target refresh rate in Hz.
- **backlight** –initial backlight percentage, from 0 to 100.

deinit()

Release the display driver resources.

width()

Return the physical display width in pixels.

height()

Return the physical display height in pixels.

clear(*display_off=...*)

Clear the display.

Parameters

display_off –True may turn the panel output off when supported by the board.

backlight(*value=...*)

Get or set backlight brightness.

Parameters

value –None returns current brightness; otherwise set 0 to 100 percent.

write(*image*, *, *x=...*, *y=...*, *x_scale=...*, *y_scale=...*, *roi=...*, *fit=...*)

Draw an image on the display. x, y: destination top-left position in display pixels. x_scale, y_scale: optional manual scale factors.

Parameters

- **image** –source image, normally from sensor.snapshot().
- **roi** –optional source rectangle (x, y, w, h).
- **fit** –True scales the image to fit the display area.

5.5 espdl –Model Inference

The `espdl` module runs ESP-DL `.espdl` models on captured images. It provides task-specific wrappers for object detection (`ESPDet`, `YOLO11`), pose estimation (`YOLO11nPose`), and image classification (`ImageNetCls`).

5.5.1 Object Detection

```
import sensor, espdl

det = espdl.ESPDet("/sdcard/espdet_pico_224_224_face.espdl", score=0.5, nms=0.7)
try:
    img = sensor.snapshot()
    for x, y, w, h, score, category in det.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0), thickness=2)
        img.draw_string(x, max(0, y - 12), "%.2f:%d" % (score, category))
finally:
    det.deinit()
```

`score` removes low-confidence candidates, while `nms` controls suppression of overlapping boxes. Detection coordinates are mapped back to the input image and can be used directly for drawing. Load a model once and reuse it across frames; `deinit()` releases model weights and intermediate buffers.

5.5.2 Image Classification

```
import espdl, image

img = image.Image("/sdcard/cat.jpg").to_rgb565(copy=True)
classifier = espdl.ImageNetCls(
    "/sdcard/imagenet_cls_mobilenetv2_s8_v1.espdl",
    topk=5,
    score=0.0,
)
try:
    for label, score in classifier.classify(img):
        print(label, "%.4f" % score)
finally:
    classifier.deinit()
```

Classification returns up to `topk (label, score)` pairs ordered by the model output. Convert file images to RGB565 when necessary so the input format is accepted by the wrapper.

5.5.3 Pose Estimation

```
pose = espdl.YOLO11nPose("/sdcard/espdet_yolo11n_pose_160_160_coco.espdl", score=0.
↪35, topk=5)
try:
    img = sensor.snapshot()
    for x, y, w, h, score, category, keypoints in pose.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
        for px, py in keypoints:
            if px > 0 and py > 0:
                img.draw_circle(px, py, 2, color=(0, 255, 0), fill=True)
finally:
    pose.deinit()
```

Each pose result contains 17 COCO keypoints. A missing or low-confidence point is returned as `(0, 0)` and should be skipped before drawing or computing joint geometry.

5.5.4 Adjust Thresholds at Runtime

```
det.set_thresholds(score=0.65, nms=0.6)
```

Thresholds can be changed without reloading the model, which is useful when adapting an application to lighting, distance, or scene-density changes.

5.5.5 Result tuples

- Detection: `(x, y, w, h, score, category)`
- Pose: `(x, y, w, h, score, category, keypoints)` with 17 COCO keypoints
- Classification: `(label, score)`

See also:

AI Inference describes the ESP-DL inference pipeline, the `.espdl` format, quantization, and pre-/post-processing. To deploy a new model, see *Introduce a New Model*.

Runnable examples: `example/03-Machine-Learning/00-ESP-DL` (ESPDet, YOLO11, pose, ImageNet classification).

5.5.6 Functions

`espdl.`**`load_model`**(*path*, *, *profile=...*)

Preload an ESP-DL model file.

Parameters

- **`path`** –model path, for example "/sdcard/model.espdl" or "/flash/model.espdl".
- **`profile`** –True enables ESP-DL profiling output when supported.

5.5.7 Classes

`class` `espdl.`**`ESPDet`**(*path*, *, *score=...*, *nms=...*, *mean=...*, *std=...*)

ESP-DL object detection wrapper for ESPDet models.

Create a detector from an .espdl model.

Parameters

- **`path`** –model path.
- **`score`** –optional confidence threshold.
- **`nms`** –optional non-maximum suppression threshold.
- **`mean`** –optional RGB mean values for preprocessing.
- **`std`** –optional RGB standard deviation values for preprocessing.

`deinit`()

Release model resources.

`detect`(*image*)

Run object detection on an image.

Parameters

`image` –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**YOLO11**(*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11 object detection wrapper.

Create a YOLO11 detector from an .espdl model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **topk** –maximum number of detections returned per frame.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run object detection on an image.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**YOLO11nPose**(*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11n COCO pose wrapper.

Create a YOLO11n pose detector from an .espdl model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **topk** –maximum number of pose results returned per frame.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run COCO pose detection on an image. Returns 17 COCO keypoints for each person.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**ImageNetCls**(*path*, *, *topk*=..., *score*=..., *mean*=..., *std*=..., *softmax*=...)

ESP-DL ImageNet classification wrapper.

Create a classifier from an .espdl model.

Parameters

- **path** –model path.
- **topk** –maximum number of classes returned.
- **score** –optional minimum score threshold.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.
- **softmax** –True applies softmax to output scores.

deinit()

Release model resources.

classify(*image*)

Run image classification on an image.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *topk*=..., *score*=...)

Update classifier thresholds.

Parameters

- **topk** –new maximum result count, or None to keep current value.
- **score** –new minimum score, or None to keep current value.

5.6 image.ImageIO –Image Stream

The `image.ImageIO` type records and replays sequences of images while preserving inter-frame timing. A file stream reads/writes a container on storage; a memory stream keeps frames in a pre-allocated buffer. It is exposed on the `image` module as `image.ImageIO`; the type stub lives in `stubs/imageio.pyi`.

5.6.1 Record to Storage

```
import sensor, image

stream = image.ImageIO("/sdcard/stream.bin", "w")
for _ in range(30):
    stream.write(sensor.snapshot())
stream.sync()
print("frames:", stream.count(), "bytes:", stream.size())
stream.close()
```

Each `write()` stores the image and the elapsed time since the previous frame. `sync()` flushes buffered file data without closing the stream. Always close a file stream so the container metadata and storage buffers are finalized.

5.6.2 Replay a Recorded Stream

```
import image

stream = image.ImageIO("/sdcard/stream.bin", "r")
try:
```

(continues on next page)

(continued from previous page)

```
    while True:
        img = stream.read(loop=False, pause=True)
        if img is None:
            break
        img.flush()
finally:
    stream.close()
```

`pause=True` reproduces the recorded frame timing, while `loop=False` returns `None` at the end instead of rewinding. Set `pause=False` when frames should be consumed as quickly as processing permits.

5.6.3 Use an In-Memory Stream

```
stream = image.ImageIO((320, 240, image.RGB565), 10)
for _ in range(10):
    stream.write(sensor.snapshot())

stream.seek(0)
first = stream.read(pause=False)
print("buffer bytes:", stream.buffer_size())
stream.close()
```

Memory streams avoid storage I/O and are useful for short frame histories or frame-difference workflows, but their full capacity is allocated in RAM or PSRAM when the stream is created.

See also:

Codecs and Streaming covers ImageIO alongside JPEG, H.264, and the USB CDC preview path.

Runnable example: `example/02-Image-Processing/03-Frame-Differencing/in_memory_frame_differencing.py` uses an in-memory ImageIO stream.

5.6.4 Constants

`imageio.`**`FILE_STREAM`**

Stream backed by a file on storage (e.g. `/sdcard/stream.bin`).

`imageio.`**`MEMORY_STREAM`**

Stream backed by a fixed-size buffer in PSRAM/RAM.

5.6.5 Classes

`class` `imageio.`**`ImageIO`**(*stream*, *mode*)

`class` `imageio.`**`ImageIO`**(*stream*, *count*)

Record and replay sequences of images, preserving inter-frame timing. Exposed as `image.ImageIO`. A file stream reads/writes the OpenMV "OMV IMG STR" container on storage; a memory stream keeps frames in a pre-allocated buffer for fast capture/playback.

Open a file stream for reading ("r") or writing ("w").

Parameters

- **`stream`** –path to the stream file.
- **`mode`** –"r" to read, "w" to create/overwrite.

`type`()

Return the stream type: FILE_STREAM or MEMORY_STREAM.

is_closed()

Return True if the stream has been closed.

count()

Return the number of frames recorded in the stream.

offset()

Return the current frame index (read/write cursor).

version()

Return the container version for file streams, or None for memory streams.

buffer_size()

Return the per-frame buffer size for memory streams, or None for file streams.

size()

Return the total stream size in bytes.

write(*image*)

Append one image to the stream (records the elapsed time since the last write).

Parameters

image –frame to store.

read(*copy_to_fb=...*, ***, *loop=...*, *pause=...*)

Read the next frame from the stream.

Parameters

- **copy_to_fb** –True loads the frame into the frame buffer (and updates the preview).
- **loop** –True rewinds to the first frame at end-of-stream instead of returning None.
- **pause** –True sleeps to honor the frame's recorded timestamp (real-time playback).

seek(*offset*)

Move the read/write cursor to the given frame index.

Parameters

offset –target frame index.

sync()

Flush buffered file data to storage (no-op for memory streams).

close()

Close the stream and release its resources.

5.7 h264 –H.264 Encoding

The `h264` module encodes images into Annex-B H.264 NAL units, for example to record video to storage or to feed the *rtsp –RTSP Streaming* server. It is available on ESP32-P4 builds.

5.7.1 Encode One Frame

```
import sensor, h264

enc = h264.H264Encoder(320, 240, fps=15)
nal = enc.encode(sensor.snapshot())
print("bytes:", len(nal), "keyframe:", enc.keyframe())
enc.close()
```

The encoder dimensions must match every input image. `encode()` returns one encoded frame as `bytes` and `keyframe()` reports whether the most recently encoded frame is an IDR/I frame.

5.7.2 Record a Raw H.264 Stream

```
import sensor, h264

FRAME_COUNT = 300
OUT_PATH = "/sdcard/out.h264"

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

first = sensor.snapshot()
enc = h264.H264Encoder(
    first.width(),
    first.height(),
    fps=15,
    bitrate=1_500_000,
    gop=15,
)

try:
    with open(OUT_PATH, "wb") as output:
        output.write(enc.encode(first))
        for _ in range(FRAME_COUNT - 1):
            output.write(enc.encode(sensor.snapshot()))
finally:
    enc.close()
```

This writes an elementary Annex-B stream without an MP4 container. It can be played with `ffplay out.h264` or remuxed on a host. `bitrate` is the target number of bits per second, `gop` is the keyframe interval in frames, and the QP bounds control the permitted quality/size range.

5.7.3 Resource and Throughput Considerations

Create one encoder for a fixed resolution and reuse it throughout the stream. Close it when finished. If capture or storage cannot sustain the configured frame rate and bitrate, reduce the resolution, frame rate, or bitrate instead of repeatedly recreating the encoder.

See also:

Codecs and Streaming explains the encode-then-stream pipeline and how H.264 keyframes interact with RTSP delivery.

Runnable example: `example/01-Camera/01-H264/record_h264.py`.

5.7.4 Classes

class `h264.`**H264Encoder**(*width*, *height*, *, *fps=...*, *gop=...*, *bitrate=...*, *qp_min=...*, *qp_max=...*)

Hardware-accelerated H.264 video encoder. Feed it images frame by frame with encode(); it returns Annex-B NAL units ready to mux into a file or stream over the network (see the rtsp module).

Open an encoder for a fixed frame size.

Parameters

- **width** –frame width in pixels.

- **height** –frame height in pixels.
- **fps** –target frame rate; also the default GOP length.
- **gop** –keyframe (IDR) interval in frames; 0 selects one keyframe per second (== fps).
- **bitrate** –target bitrate in bits per second; 0 auto-selects width*height*fps.
- **qp_min** –lower quantization-parameter bound (better quality, larger frames).
- **qp_max** –upper quantization-parameter bound (lower quality, smaller frames).

encode(*image*)

Encode one image and return its Annex-B NAL units as bytes.

Parameters

image –source frame; its size must match the encoder width/height.

keyframe()

Return True if the most recently encoded frame was a keyframe (IDR/I).

close()

Release the encoder. Using the object afterwards raises OSError.

5.8 rtsp –RTSP Streaming

The `rtsp` module serves H.264 NAL units over RTSP so a client such as VLC or ffplay can view the camera stream over the network. Pair it with *h264 –H.264 Encoding*. It is available on ESP32-P4 builds.

5.8.1 Capture, Encode, and Stream

```
import network, sensor, h264, rtsp, time
from machine import Pin

WIDTH, HEIGHT, FPS = 320, 240, 30

lan = network.LAN(
    mdc=Pin(31),
    mdio=Pin(52),
    reset=Pin(51),
    phy_addr=1,
    phy_type=network.PHY_IP101,
)
lan.active(True)

for _ in range(100):
    if lan.isconnected():
        break
    time.sleep_ms(100)
if not lan.isconnected():
    raise OSError("Ethernet connection failed")

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

enc = h264.H264Encoder(WIDTH, HEIGHT, fps=FPS, bitrate=3_000_000)
server = rtsp.RTSPServer(WIDTH, HEIGHT, fps=FPS, listen_port=8554)

print("rtsp://%s:8554/" % lan.ifconfig()[0])
try:
```

(continues on next page)

(continued from previous page)

```
    while True:
        server.send(enc.encode(sensor.snapshot()))
finally:
    server.stop()
    enc.close()
```

The Ethernet pin mapping above is for the ESP32-P4X-Function-EV-Board with its IP101 RMII PHY; use the network configuration appropriate for another product board. The width, height, and frame rate advertised by `RTSPServer` must match the encoder configuration.

5.8.2 Client and Queue Behavior

Open the printed URL with `ffplay rtsp://<board-ip>:8554/` or an RTSP-capable player. `send()` queues one encoded frame and does not wait for a client; when no client is connected or the client is too slow, frames may be dropped so the capture loop is not blocked. `max_frame_len` can be increased when high-resolution or high-bitrate encoded frames exceed the default limit.

5.8.3 Shutdown

Always call `server.stop()` before releasing network or encoder resources. A `try/finally` block ensures cleanup also occurs when the script is interrupted or an exception is raised.

See also:

Codecs and Streaming covers the full capture, encode, and stream pipeline.

Runnable example: `example/01-Camera/02-RTSP/stream_rtsp.py`.

5.8.4 Classes

class rtsp.**RTSPServer**(*width*, *height*, *, *fps=...*, *listen_port=...*, *max_frame_len=...*)

Minimal RTSP server that streams H.264 video over the network. Create it once Wi-Fi is up, then push encoded NAL units from an h264.H264Encoder with send(). Clients connect to `rtsp://<board-ip>:<listen_port>/`. Only video (no audio) is served.

Start the RTSP server and advertise the given stream format.

Parameters

- **width** –advertised video width in pixels.
- **height** –advertised video height in pixels.
- **fps** –advertised frame rate, used for SDP and pacing.
- **listen_port** –TCP port the RTSP service binds to.
- **max_frame_len** –max accepted encoded frame size in bytes; 0 uses width*height as a safe ceiling.

send(*nal*)

Queue one encoded H.264 frame (Annex-B NAL units) for delivery. Frames are sent in order; if the client is slow or absent the frame is dropped rather than blocking the caller. Frames larger than max_frame_len are ignored.

Parameters

nal –encoded frame bytes from h264.H264Encoder.encode().

stop()

Stop the server, join its task, and free queued frames.

Chapter 6

How-To Guides

Task-oriented guides for extending ESP-VISION.

6.1 Add a New Python Module

ESP-VISION exposes Python modules through the MicroPython `USER_C_MODULES` mechanism. The binding layer lives in `modules/` and only does object conversion plus light API adaptation; heavy logic belongs in pure C or in `platform/`. This guide adds a module named `foo`.

6.1.1 Overview

```
modules/py_foo.c            # binding + MP_REGISTER_MODULE(MP_QSTR_foo, ...)
modules/qstrdefs_esp_vision.h   # any feature-gated qstrs
micropython.cmake           # add py_foo.c to the source list
stubs/foo.pyi               # type stub for IDE completion
docs/.../api-reference/foo.rst  # reference page
```

6.1.2 1. Create the Binding Source

Add `modules/py_foo.c`. Define the module's functions, build a globals dict, and self-register the module. Existing modules are the best templates; for a simple function-only module follow `modules/py_sensor.c`, and for a type/class module follow `modules/py_display.c`.

```c
/*
 * SPDX-FileCopyrightText: 2026 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#include "py/runtime.h"

static mp_obj_t foo_hello(void) {
    return mp_obj_new_int(42);
}
static MP_DEFINE_CONST_FUN_OBJ_0(foo_hello_obj, foo_hello);

static const mp_rom_map_elem_t foo_module_globals_table[] = {
```

(continues on next page)

(continued from previous page)

```
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_foo) },
    { MP_ROM_QSTR(MP_QSTR_hello), MP_ROM_PTR(&foo_hello_obj) },
};
static MP_DEFINE_CONST_DICT(foo_module_globals, foo_module_globals_table);

const mp_obj_module_t mp_module_foo = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&foo_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_foo, mp_module_foo);
```

C++ modules use `.cpp` (see `modules/py_espdl.cpp`); the build already sets `-std=gnu++2b` for C++ sources.

6.1.3 2. Register qstrs If Needed

Most qstrs (`MP_QSTR_foo`, `MP_QSTR_hello`) are collected automatically from the source. Only add entries to `modules/qstrdefs_esp_vision.h` for names that are board-feature-gated or otherwise not seen by the qstr scanner, using the `Q(name)` form.

6.1.4 3. Wire the Source into the Build

Add the source to `ESP_VISION_MODULE_SOURCES` in `micropython.cmake`:

```
set(ESP_VISION_MODULE_SOURCES
    ${ESP_VISION_ROOT}/modules/py_display.c
    ${ESP_VISION_ROOT}/modules/py_image.c
    ${ESP_VISION_ROOT}/modules/py_foo.c    # new
    ...
)
```

If the module is only valid on some chips, add it inside the matching `IDF_TARGET` block (the H.264 and RTSP modules are gated to `esp32p4` this way). If it needs an extra component, link it via `target_link_libraries` like `idf::zxing`.

6.1.5 4. Add a Type Stub

Create `stubs/foo.pyi` describing the public surface so IDEs can complete it. Keep the Apache SPDX header and match the style of the existing stubs.

6.1.6 5. Document the Module

Add `docs/en/api-reference/foo.rst` and `docs/zh_CN/api-reference/foo.rst` using the Python-domain directives (`.. py:module::`, `.. py:function::`, `.. py:class::`), and add `foo` to the `toctree` in `api-reference/index.rst` for both languages.

6.1.7 6. Build and Verify

```
idf.py --board ESP32_P4X_EYE build
```

Then check from the REPL:

```
import foo
print(foo.hello())
```

6.2 Customize Firmware Features

ESP-VISION separates application-facing modules, image algorithms, standard MicroPython features, frozen Python code, and board services into independent configuration layers. This lets a product firmware remove unused capabilities to reduce flash, RAM, dependencies, and attack surface, or add product-specific modules without rewriting the camera and media pipeline.

6.2.1 Choose the Customization Scope

Before editing configuration, decide whether the change belongs to every ESP-VISION firmware or only one board. Changes to the root `micropython.cmake` affect every board whose chip matches the edited condition. Changes under `boards/<BOARD>/` (including the MicroPython-port files in `boards/<BOARD>/port/`) are board-specific. For a product variant, create a dedicated board package as described in *Add a New Board* instead of changing a shared development-board profile.

6.2.2 Customize ESP-VISION Python Modules

The `ESP_VISION_MODULE_SOURCES` list in `micropython.cmake` defines the C/C++ bindings exposed to Python. Remove a source only together with any helper source it requires, or append a new source as described in *Add a New Python Module*. Chip-specific modules belong inside an `IDF_TARGET` condition; for example, H.264 and RTSP are currently enabled only for `esp32p4`.

```
set(ESP_VISION_MODULE_SOURCES
    ${ESP_VISION_ROOT}/modules/py_display.c
    ${ESP_VISION_ROOT}/modules/py_image.c
    ${ESP_VISION_ROOT}/modules/py_imageio.c
    ${ESP_VISION_ROOT}/modules/py_helper.c
    ${ESP_VISION_ROOT}/modules/py_sensor.c
)
```

Removing a binding does not automatically remove every platform service or managed component behind it. After changing the source list, inspect `target_sources`, `target_link_libraries`, component manifests, and the final size report to confirm that the unwanted dependency is no longer linked.

6.2.3 Customize Image Algorithms

Each board's `boards/<BOARD>/imlib_config.h` selects optional `imlib` algorithms. Remove an `IMLIB_ENABLE_*` definition to omit an unused algorithm, or add a supported definition to expose another algorithm. Common groups include filters, geometry detection, QR codes, and AprilTags.

```
#define IMLIB_ENABLE_MEAN
#define IMLIB_ENABLE_GAUSSIAN
#define IMLIB_ENABLE_QRCODES
#define IMLIB_ENABLE_APRILTAGS
```

Some Python methods remain present in the binding even when their backend is disabled and will raise an exception at runtime. Keep the API documentation, examples, and generated board support table aligned with the selected algorithms. Do not remove `OMV_NO_GPL` without completing a license review.

6.2.4 Customize Standard MicroPython Features

Use `boards/<BOARD>/port/mpconfigboard.h` to override standard MicroPython feature macros for one board, such as networking, Bluetooth, ESP-NOW, ADC, SD card, USB, or other `MICROPY_PY_*` and `MICROPY_HW_*` options. Use `mpconfigboard.cmake` for CMake-level options and the board's sdkconfig chain.

```
#define MICROPY_PY_BLUETOOTH (0)
#define MICROPY_PY_ESPNOW (0)
#define MICROPY_PY_NETWORK_WLAN (1)
```

These macros can depend on ESP-IDF versions and SoC capability macros. A disabled Python feature may also require removing its ESP-IDF configuration or managed dependency before it produces a measurable firmware-size reduction.

6.2.5 Customize Frozen Python Code

The board's `boards/<BOARD>/manifest.py` controls Python modules frozen into firmware. Add product startup code, libraries, or packages with `freeze()`, `module()`, `package()`, or `include()` as supported by the MicroPython manifest system. Remove an entry when that frozen module is not required.

```
freeze("$(PORT_DIR)/modules")
freeze("$(ESP_VISION_ROOT)/modules", "py_inisetup.py")
freeze("$(ESP_VISION_ROOT)/boards/<BOARD>", "board_inisetup.py")
include("$(MPY_DIR)/extmod/asyncio")
```

Freezing code improves deployment consistency and startup availability but consumes firmware flash. Files intended to remain replaceable during development or after deployment should stay on the root filesystem, such as packages under `/lib`, or on `/sdcard` instead.

6.2.6 Customize Board Services and Optional Components

Board-specific camera, display, and SD card implementations live under `boards/<BOARD>/`. `micropython.cmake` automatically selects `camera.c`, `display.c`, and `sdcard.c` when present. Use `boards/<BOARD>/board.cmake` for board-level CMake switches; for example, current P4 boards enable the optional ZXing barcode backend with `ESP_VISION_ENABLE_BARCODE`.

```
set(ESP_VISION_ENABLE_BARCODE OFF)
```

When adding a component, register its source or IDF component and link it to `usermod_esp_vision_platform`. When removing one, remove the source, include path, compile definition, link dependency, and component-manifest entry as a single change.

6.2.7 Build and Verify

Reconfigure after changing CMake, sdkconfig, manifests, or feature macros, then build the selected board:

```
idf.py --board <BOARD> reconfigure
idf.py --board <BOARD> build
idf.py --board <BOARD> size
```

Verify imports and affected APIs on the REPL, run the relevant examples, and compare the size report before and after the change. Also rebuild the documentation because module navigation and board capability summaries are generated from the firmware configuration.

6.3 Introduce a New Model

ESP-VISION runs ESP-DL `.espdl` models through the *espdl – Model Inference* module. Models are not built into the firmware; they live on board storage and are loaded at runtime. This guide adds a new model and runs it.

6.3.1 1. Obtain or Convert the Model

Get a ready `.espdl` from the ESP-DL model zoo, or convert your own model to the `.espdl` format with the ESP-DL quantization/export toolchain, matching the selected chip (ESP32-P4, ESP32-S3, or ESP32-S31).

Keep the directory layout under `models/` in the repository when adding shared assets, mirroring `models/espdet/`.

6.3.2 2. Copy the Model to Board Storage

Place the `.espdl` file on storage the firmware can read, such as `/sdcard` or `/flash`:

- SD card: copy the file onto the card, which mounts at `/sdcard`.
- On-flash FAT (`ffat`): the data partition is exposed over USB MSC, so you can drag the file onto the mass-storage drive; it is visible as `/flash`.

6.3.3 3. Pick the Right Wrapper

Choose the `espdl` wrapper that matches the model task:

Task	Class	Result
Object detection (ESPDet)	*espdl.ESPDet*	`(x, y, w, h, score, category)`
Object detection (YOLO11)	*espdl.YOLO11*	`(x, y, w, h, score, category)`
Pose detection	*espdl.YOLO11nPose*	detection plus 17 COCO keypoints
Image classification	*espdl.ImageNetCls*	`(label, score)`

If the model needs different preprocessing, pass `mean`, `std`, `score`, `nms`, `topk`, or `softmax` to the constructor.

6.3.4 4. Run Inference

```
import sensor, image, espdl

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)

det = espdl.ESPDet("/sdcard/my_model.espdl", score=0.5, nms=0.45)

while True:
    img = sensor.snapshot()
    for x, y, w, h, score, category in det.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
    img.flush()
```

See `example/03-Machine-Learning/00-ESP-DL/` for runnable scripts (`espdet_pico.py`, `yolo11.py`, `yolo11n_pose.py`, `imagenet_cls.py`).

6.3.5 5. Optional: Profiling

Use *espdl.load_model()* with `profile=True` to emit ESP-DL profiling output when verifying a new model's performance.

6.4 Add a New Board

A board package lives in a single tree, `boards/<BOARD>/`: the ESP-VISION files at the top level and the MicroPython ESP32 port files under the `boards/<BOARD>/port/` subdirectory. The build projects `boards/<BOARD>/port/` onto `ports/esp32/boards/<BOARD>/` of the generated MicroPython copy. Start from the `TEMPLATE` board.

6.4.1 MicroPython Port Side

In `boards/<BOARD>/port/`:

File	Purpose
`mpconfigboard.cmake`	`IDF_TARGET` value and `SDKCONFIG_DEFAULTS` chain.
`mpconfigboard.h`	MicroPython feature flags and USB strings.
`sdkconfig.board`	Board-specific ESP-IDF Kconfig overrides.
`partitions-*.csv`	Partition table.
`board.json`, `board.md`	Upstream board manifest metadata.

6.4.2 ESP-VISION Side

In `boards/<BOARD>/`:

File	Purpose
`boardconfig.h`	Pin assignments and board runtime constants.
`imlib_config.h`	OpenMV `imlib` feature switches.
`manifest.py`	Frozen Python modules.
`camera.c`	Board-specific camera backend.
`display.c`	LCD panel and backlight implementation.
`sdcard.c`	SD card power and card-detect implementation.

`micropython.cmake` automatically picks up `camera.c`, `display.c`, and `sdcard.c` from the board directory when present, and includes the board's optional `board.cmake`.

6.4.3 Build and Flash

```
idf.py --board <NEW_BOARD> -p /dev/ttyACM0 build flash monitor
```

Note: This page is a starting outline. Detailed bring-up steps (sensor selection, PPA configuration, display timing) will be expanded.

Chapter 7

Solution Architecture

ESP-VISION is organized around a MicroPython firmware build, board-specific hardware backends, shared platform services, and Python-facing vision modules. Code is layered by whether it touches MicroPython (`mp_obj_t` / `py/*.h`).

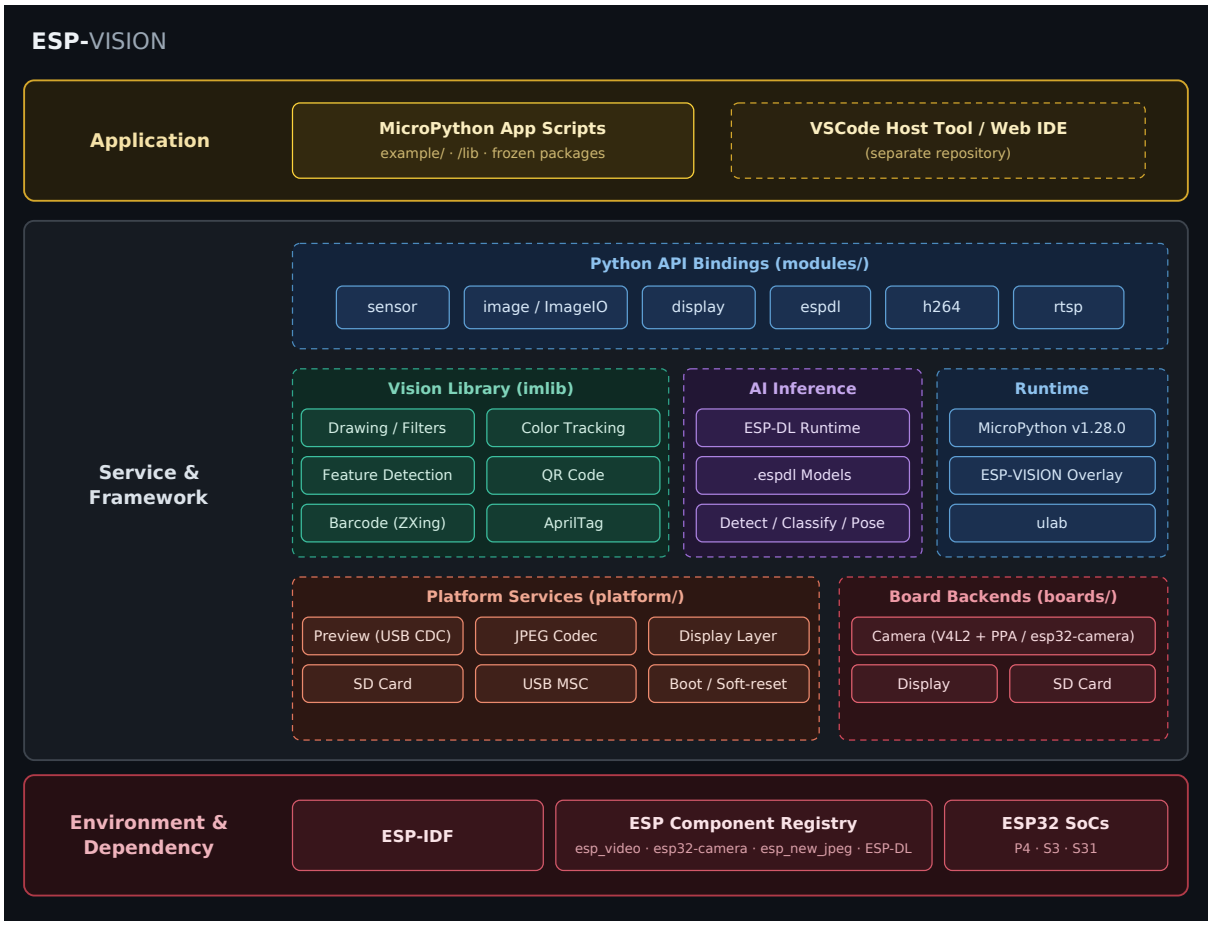
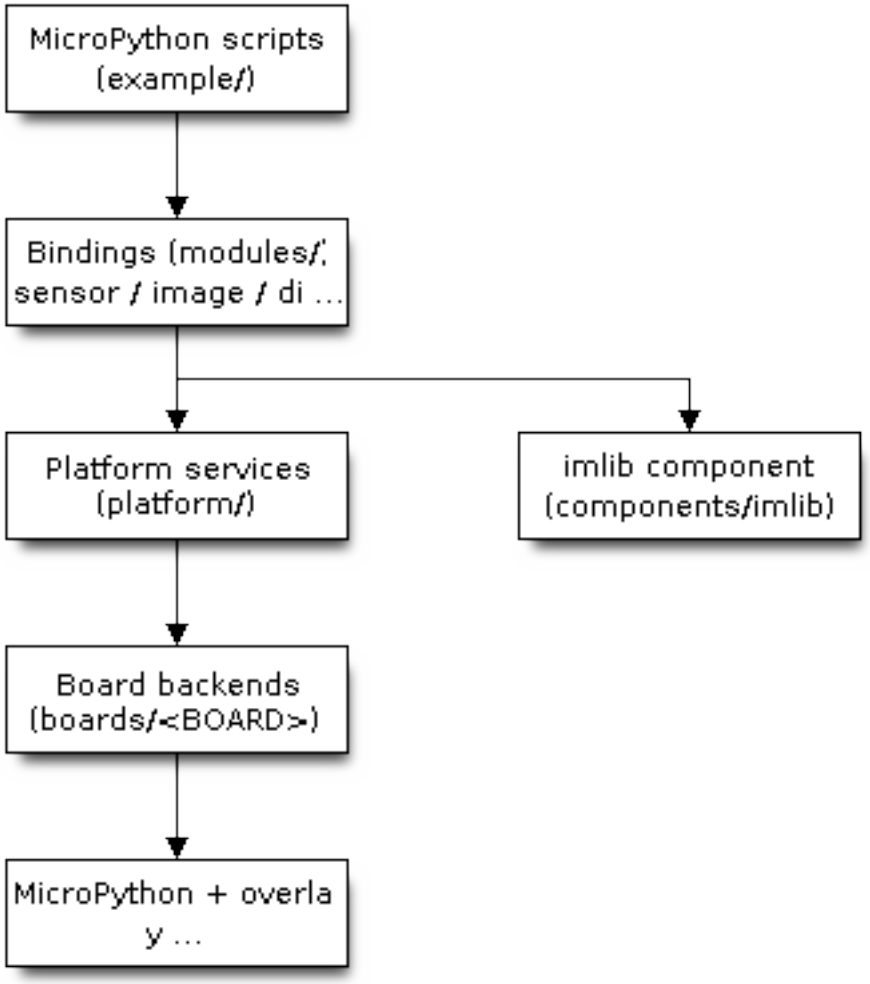


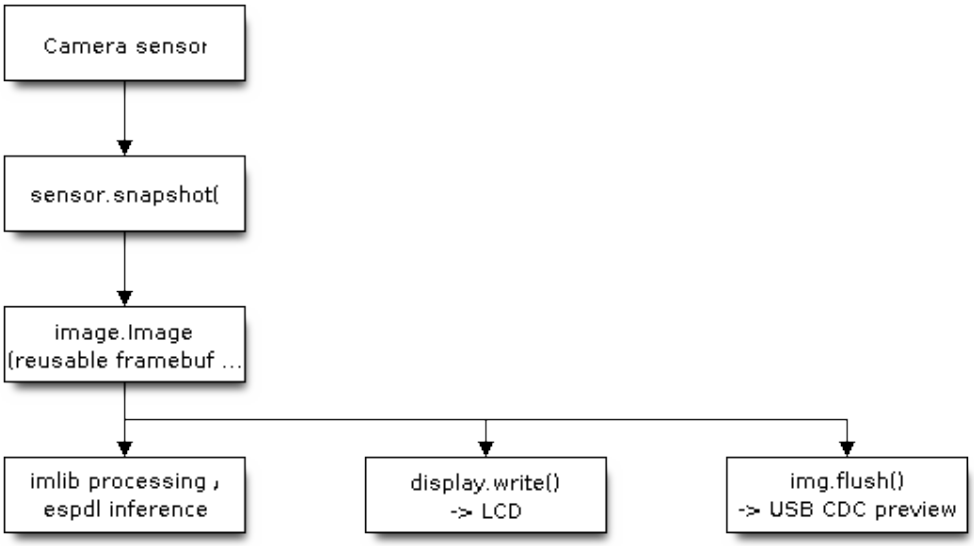
Fig. 1: ESP-VISION layered architecture overview

7.1 Layered Overview



- **Bindings** (`modules/`): the `USER_C_MODULES` layer. The four real modules `image`, `sensor`, `display`, and `espdl` self-register via `MP_REGISTER_MODULE`. `py_imageio.c` provides the `image.ImageIO` type, and `py_helper.c` is shared helper code. Bindings only do object conversion and light API adaptation; heavy logic lives in pure C or `platform/`.
- **Platform services** (`platform/`): self-written ESP32 glue. `preview.c` (EVFRAME JPEG preview over USB CDC), `display.c` (generic display layer), `sdcard.c` (mount at `/sdcard`), `usb_msc.c` (exposes the `ffat` partition over TinyUSB MSC), `jpeg.c` (hardware or software JPEG), `debug.c`, and `main.c` (startup init plus the soft-reset loop).
- **imlib component** (`components/imlib/`): pure-C vision algorithms, an IDF component maintained as MIT, derived from OpenMV `lib/imlib`.
- **Board backends** (`boards/<BOARD>/`): per-board configuration and the real camera/display/sdcard implementations. P4X and S31 use `esp_video`/V4L2; P4X also uses PPA, while S3 uses `esp32-camera`.
- **MicroPython + overlay**: MicroPython v1.28.0 is the fixed baseline; project changes live in `overlay/micropython/` and are applied to a generated build copy under `build/micropython/`.

7.2 Capture-to-Output Data Flow



`sensor.snapshot()` captures a frame into a reusable framebuffer wrapped as an `image.Image`. Scripts then run `imlib` processing or `espdl` inference on the image, and send it to the LCD with `display.write()` or to the host preview with `img.flush()`.

7.3 Source Tree

Path	Responsibility
`idf_ext.py`	Board-aware `idf.py` extension for the repository root.
`micropython.cmake`	Integration hub: registers user modules, platform and board sources, include paths, conditional `zxing`, and `ulab`.
`lib/`	Pinned third-party submodules (MicroPython, `ulab`, ZXing-C++).
`overlay/micropython/`	ESP-VISION MicroPython delta, using the MicroPython path layout.
`boards/`	Per-board config, frozen manifests, and board peripheral backends.
`platform/`	Shared runtime services (camera, preview, storage, display, USB, JPEG).
`modules/`	MicroPython C/C++ bindings (`sensor`, `image`, `display`, `imageio`, `espdl`, plus chip-dependent `h264` and `rtsp`).
`components/`	ESP-IDF components, including OpenMV `imlib` and the ZXing backend.
`models/`	Optional `.espdl` model assets loaded from board storage at runtime.
`example/`	MicroPython example scripts.
`stubs/`	`.pyi` type stubs describing the C modules.

7.4 Board Composition

A board is defined in a single tree, `boards/<BOARD>/`:

- ESP-VISION side (top level): `boardconfig.h`, `imlib_config.h`, `manifest.py`, and optional `camera.c` / `display.c` / `sdcard.c`.

- MicroPython port side (`boards/<BOARD>/port/`): `IDF_TARGET` value, sdkconfig, partitions, and USB strings. The build projects this subdirectory onto `ports/esp32/boards/<BOARD>/` of the generated MicroPython copy.

See *Add a New Board* for the step-by-step procedure.

7.5 Chip-Dependent Sources

`micropython.cmake` selects modules from `IDF_TARGET` and the board profile. The ESP32-P4 build includes `h264` and `rtsp`; the current P4 board profiles also enable the ZXing-C++ barcode backend. See *Chip and Board Support* for the resulting public API matrix.

7.6 MicroPython Overlay

ESP-VISION uses MicroPython v1.28.0 as a fixed upstream baseline. Project changes to the ESP32 port live under `overlay/micropython/`. The `prepare-micropython` build step applies that tree to a generated copy under `build/micropython/idf<ESP_IDF_VERSION>/micropython/`; the `lib/micropython` submodule remains a clean upstream reference.

For how ESP-VISION relates to its upstream projects, see *Project Relationships*; for the license of each component, see *Licensing*.

Chapter 8

Project Relationships

This chapter explains how ESP-VISION relates to MicroPython, OpenMV, and its major third-party components. For licensing and redistribution obligations, see *Licensing*.

8.1 ESP-VISION, MicroPython, and OpenMV

The relationship can be summarized as follows:

- **MicroPython provides the language runtime and firmware base.** ESP-VISION starts from a pinned MicroPython ESP32 port and adds `overlay/micropython/`, MicroPython user C modules, and ESP-IDF components. It does not implement a separate Python interpreter.
- **OpenMV is an upstream source for part of the vision implementation and API design.** ESP-VISION reuses a subset of OpenMV `imlib` algorithms and some `image` binding and helper code, preserving their original license and copyright notices.
- **ESP-VISION is an independent integration for Espressif chips.** Camera, display, storage, USB, codec, ESP-DL inference, and board support are integrated using ESP-VISION and ESP-IDF components. ESP-VISION is not a fork of the OpenMV firmware and does not include the complete OpenMV hardware abstraction, IDE, or feature set.

The `sensor` and `image` modules therefore follow the OpenMV API style to make script migration easier, but **matching API names do not imply complete behavioral or feature compatibility**. Support also depends on the chip, board, memory, peripherals, and build options. Use this guide's *API Reference* as the authoritative support reference.

8.2 Dependency Layers

The main code layers in an ESP-VISION firmware image are:

```
User scripts
   |
   +-- MicroPython runtime and ESP32 port
   |
   +-- ESP-VISION Python modules
   |      +-- OpenMV-style sensor / image APIs
   |      +-- display / imageio / espdl / h264 / rtsp
   |
   +-- Algorithms and middleware
   |      +-- OpenMV imlib subset
   |      +-- ulab / ZXing-C++ / ESP-DL
```

(continues on next page)

(continued from previous page)

```
   |
   +-- ESP-IDF, managed components, and board backends
```

Dependencies come from three locations:

- Git submodules under `lib/`, such as MicroPython, `ulab`, and ZXing-C++;
- third-party-derived code maintained under `components/` and parts of `modules/`;
- ESP Component Registry packages resolved from `idf_component.yml`. The exact component set and versions can vary with the ESP-IDF version, selected chip, and board.

For the version, local path, and license of each dependency, see the *Licensing* inventory.

Chapter 9

Licensing

This chapter is the single reference for how licensing works across ESP-VISION. It is developer guidance, not legal advice; the license text and source-file headers for the exact version in use remain authoritative.

ESP-VISION' s own code (`platform/`, most of `modules/`, `boards/`, and build files) is released under Apache License 2.0, recorded in the repository-level `LICENSE`. Vendored and third-party code keeps the license declared by its own source files, SPDX identifiers, or upstream license; the repository-level `LICENSE` does not override or replace those.

9.1 License Inventory

The following table lists the direct dependencies most relevant to redistribution. It is not a complete inventory of transitive dependencies.

Project or code	Local path	Role in ESP-VISION	License
MicroPython v1.28.0	`lib/micropython`	Python runtime and ESP32 port baseline	MIT
micropython-ulab 6.12	`lib/ulab`	Array and numerical computing	MIT
OpenMV `imlib` subset	`components/imlib`	Image processing and drawing algorithms	MIT, with separately licensed files
OpenMV Python binding code	`modules/py_image*`, `modules/py_helper*`	OpenMV-style `image` API bindings	MIT
OpenMV AprilTag implementation	`components/imlib/upstream/apriltag.c`	AprilTag and rectangle detection	BSD-2-Clause
ZXing-C++ v3.0.2	`lib/zxing-cpp`	1D barcode reader backend	Apache-2.0
ESP-DL	Component Registry	Model inference runtime	MIT
esp_new_jpeg	Component Registry	Software JPEG codec	Espressif MIT (product-scope)
esp32-camera	Component Registry	Camera driver	Apache-2.0
ESP-IDF	External SDK	Build system, drivers, and media components	Apache-2.0

9.2 How to Interpret the Licenses

The repository-level `LICENSE` covers original ESP-VISION code and does not override third-party licenses:

- third-party files remain under their original licenses and retain their original copyright and license headers;
- a directory can contain multiple licenses: most `imlib` code is MIT while `apriltag.c` is BSD-2-Clause;
- permissive licenses such as Apache-2.0, MIT, and BSD generally allow combined distribution, but their copyright, license text, and notice requirements still apply;
- some Espressif components use licenses with product-scope conditions. For example, the current `esp_new_jpeg` license text limits use to Espressif products, so it should not be described only as unrestricted generic MIT;
- trademarks, patents, model files, and datasets may have separate terms that cannot be inferred from the source-code license.

Before distributing source or firmware, check at least:

1. the repository-level `LICENSE`;
2. the `LICENSE` in every Git submodule;
3. `managed_components/*/LICENSE` from the selected build;
4. SPDX identifiers, license headers, and copyright notices in modified or newly added source files;
5. provenance and terms for non-code assets such as models, fonts, and test images.

9.3 OpenMV and GPL Code Paths

Not every optional OpenMV upstream algorithm uses the same license. ESP-VISION sets `OMV_NO_GPL=1` in `micropython.cmake` and `components/imlib/CMakeLists.txt` so GPL-conditioned OpenMV code paths are not compiled. This switch only describes what the default build excludes; it is not an automatic license classification for newly added files.

When changing `imlib` feature switches, synchronizing OpenMV upstream code, or adding algorithms, inspect each file's license. Do not infer a license only from the directory containing the file.

9.4 Adding Third-Party Code

When adding a third-party package or source file to ESP-VISION:

1. pin a reproducible release or commit;
2. verify the license per file and its compatibility with the intended distribution;
3. preserve upstream copyright, SPDX identifiers, license text, and any required NOTICE;
4. record the source, version, local path, purpose, and license in this inventory;
5. verify that build switches actually exclude code that is not intended for distribution instead of relying on documentation alone;
6. obtain legal review before merging when licensing is unclear or imposes commercial-distribution constraints.

Python Module Index

Index

E

F

G

H

I

J

K

L

M

O

P

Q

R

S

T

U

V

W

X

Y