

ESP32-P4

ESP-VISION Programming Guide



Release 2026.07.28-18-g3a88cb4321
Espressif Systems
Aug 21, 2026

Table of contents

Table of contents	i
1 Introduction	3
1.1 What Is ESP-VISION	3
1.2 Quick Experience	3
1.3 Key Features	3
1.4 Supported Boards	3
2 Chip and Board Support	5
2.1 Supported Development Boards	6
2.2 Chip Versus Board Capabilities	7
2.3 Standard MicroPython Features	7
3 Get Started	9
3.1 Prerequisites	9
3.2 Build, Flash, and Monitor	9
3.3 Common idf.py Commands	10
3.4 Run Your First Script	10
4 Concepts	11
4.1 MicroPython Language Fundamentals	11
4.1.1 Execution Model	11
4.1.2 Basic Syntax	12
4.1.3 Main Data Structures	13
4.1.4 Design Principles for Microcontrollers	14
4.1.5 MicroPython and CPython	14
4.2 Packages and Deployment	15
4.2.1 Module and Package Layout	16
4.2.2 Package Management in the Current Firmware	16
4.2.3 Packages on the Filesystem	16
4.2.4 Packages Embedded in Firmware	17
4.2.5 Choosing a Deployment Method	17
4.2.6 Import Priority and Version Control	18
4.3 The Image Model	18
4.3.1 Pixel formats	18
4.3.2 Color spaces	19
4.3.3 The frame buffer and <code>fb_alloc</code>	19
4.3.4 Region of interest (ROI)	19
4.4 Image Processing	19
4.4.1 A typical processing pipeline	19
4.4.2 Thresholding and segmentation	20
4.4.3 Neighborhood (convolution) filters	20
4.4.4 Rank and edge-preserving filters	21
4.4.5 Morphology	21
4.4.6 Histograms and statistics	21
4.4.7 Feature detection	21
4.5 AI Inference	22

4.5.1	Model Runtimes	22
4.5.2	Model Files and Quantization	22
4.5.3	Inference flow	22
4.5.4	Pre-processing	23
4.5.5	Post-processing	23
4.5.6	Memory and performance	24
4.6	The Camera Pipeline	24
4.6.1	From sensor to image	25
4.6.2	Per-board backends	25
4.6.3	Frame sizes	26
4.7	Codecs and Streaming	26
4.7.1	Still-image codecs: JPEG and PNG	26
4.7.2	Recording sequences: ImageIO	26
4.7.3	Video codec: H.264	26
4.7.4	Network streaming: RTSP	27
4.7.5	Host preview: USB CDC	27
4.8	Startup Sequence	27
4.8.1	Hard Reset and Soft Reset	28
4.8.2	ESP-VISION Startup Order	28
4.8.3	First Boot and Flash Filesystem	30
4.8.4	Using boot.py	30
4.8.5	Using main.py	30
4.8.6	REPL and Recovery	31
5	API Reference	33
5.1	MicroPython APIs	33
5.1.1	Documentation Baseline	33
5.1.2	Language and Built-in APIs	33
5.1.3	Standard and Micro Libraries	34
5.1.4	JSON and Time Example	34
5.1.5	Hardware and ESP32 APIs	34
5.1.6	GPIO and PWM Example	34
5.1.7	Networking APIs	35
5.1.8	Connect to Wi-Fi	35
5.1.9	Filesystem and Runtime APIs	35
5.1.10	List Storage and Check Memory	35
5.1.11	Structuring a Complex Application	36
5.1.12	Threads and Native Tasks	38
5.1.13	Check Availability on a Firmware	38
5.2	sensor – Camera	38
5.2.1	Camera Initialization and Continuous Capture	38
5.2.2	Image Orientation and Camera Status	39
5.2.3	Temporarily Stop Capture	39
5.2.4	Constants	39
5.2.5	Functions	40
5.2.6	Classes	41
5.3	image – Image Processing	41
5.3.1	Drawing and Color-Blob Tracking	41
5.3.2	Filtering and Binary Segmentation	41
5.3.3	QR Code Recognition	42
5.3.4	Line, Segment, and Circle Detection	42
5.3.5	Encode and Save an Image	42
5.3.6	Constants	43
5.3.7	Functions	45
5.3.8	Classes	46
5.4	display – LCD Output	58
5.4.1	Camera Preview	58
5.4.2	Position, Scale, and Crop	58

5.4.3	Backlight and Cleanup	59
5.4.4	Classes	59
5.5	espdll-Model Inference	60
5.5.1	Raw Output Tensors	60
5.5.2	Object Detection	60
5.5.3	PicoDet	60
5.5.4	PedestrianDetect	61
5.5.5	Image Classification	61
5.5.6	Pose Estimation	61
5.5.7	Adjust Thresholds at Runtime	61
5.5.8	Result tuples	62
5.5.9	Functions	62
5.5.10	Classes	62
5.6	tfllite-Model Inference	65
5.6.1	Basic Usage	65
5.6.2	Callable Inputs	65
5.6.3	Post-processing	65
5.6.4	Classes	66
5.7	image.ImageIO-Image Stream	66
5.7.1	Record to Storage	66
5.7.2	Replay a Recorded Stream	66
5.7.3	Use an In-Memory Stream	67
5.7.4	Constants	67
5.7.5	Classes	67
5.8	h264-H.264 Encoding	68
5.8.1	Encode One Frame	68
5.8.2	Record a Raw H.264 Stream	69
5.8.3	Resource and Throughput Considerations	69
5.8.4	Classes	69
5.9	rtsp-RTSP Streaming	70
5.9.1	Capture, Encode, and Stream	70
5.9.2	Client and Queue Behavior	71
5.9.3	Shutdown	71
5.9.4	Classes	71
6	How-To Guides	73
6.1	Add a New Python Module	73
6.1.1	Overview	73
6.1.2	1. Create the Binding Source	73
6.1.3	2. Register qstrs If Needed	74
6.1.4	3. Wire the Source into the Build	74
6.1.5	4. Add a Type Stub	74
6.1.6	5. Document the Module	74
6.1.7	6. Build and Verify	74
6.2	Customize Firmware Features	75
6.2.1	Choose the Customization Scope	75
6.2.2	Customize ESP-VISION Python Modules	75
6.2.3	Customize Image Algorithms	75
6.2.4	Customize Standard MicroPython Features	76
6.2.5	Customize Frozen Python Code	76
6.2.6	Customize Board Services and Optional Components	76
6.2.7	Build and Verify	76
6.3	Introduce a New Model	77
6.3.1	1. Obtain or Convert the Model	77
6.3.2	2. Copy the Model to Board Storage	77
6.3.3	3. Pick the Right API	77
6.3.4	4. Run ESP-DL Inference	77
6.3.5	5. Decode ESP-DL Outputs in Python	78

6.3.6	6. Run TFLite Micro Inference	78
6.3.7	7. Optional: Profiling and Validation	79
6.4	Train an ESPDet Pico Model	79
6.4.1	1. Prepare the ESP-Detection Environment	79
6.4.2	2. Prepare the Dataset	80
6.4.3	3. Train ESPDet Pico	80
6.4.4	4. Prepare Calibration Data	81
6.4.5	5. Quantize and Export .espd	81
6.4.6	6. Add Model Metadata	82
6.4.7	7. Copy the Model to Board Storage	82
6.4.8	8. Run on ESP-VISION	83
6.4.9	9. Validate and Tune	83
6.5	Add a New Board	84
6.5.1	MicroPython Port Side	84
6.5.2	ESP-VISION Side	84
6.5.3	Build and Flash	85
6.6	Agent Code Generation and EV-MUX Closed-Loop Test	85
6.6.1	Dependency Rules	85
6.6.2	All-in-One Output	85
6.6.3	USB-OTG CDC and USJ Transport Contract	86
7	Solution Architecture	93
7.1	Layered Overview	94
7.2	Capture-to-Output Data Flow	95
7.3	Low-Level Transport	95
7.3.1	Physical Sinks	95
7.3.2	Routed Streams	96
7.3.3	EV-MUX Channels	96
7.3.4	EV-MUX Frame Format	97
7.3.5	EV-ATP Control	97
7.3.6	Host Integration Contract	98
7.3.7	Receive-Side Demultiplexing	98
7.3.8	Transport Execution Model	98
7.3.9	IDF Console and Interface Selection	99
7.3.10	Transport Reliability and Observability	99
7.4	Source Tree	100
7.5	Board Composition	100
7.6	Chip-Dependent Sources	100
7.7	MicroPython Overlay	101
8	Project Relationships	103
8.1	ESP-VISION, MicroPython, and OpenMV	103
8.2	Dependency Layers	103
9	Licensing	105
9.1	License Inventory	105
9.2	How to Interpret the Licenses	106
9.3	OpenMV and GPL Code Paths	106
9.4	Adding Third-Party Code	106
	Python Module Index	107
	Python Module Index	107
	Index	109
	Index	109

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 Quick Experience

Visit the [ESP-VISION website](#) for quick access to [Web IDE](#), MCP setup resources, examples, and related project resources.

1.3 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, plus TensorFlow Lite Micro support for `.tflite` model execution.
- 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`.

1.4 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.

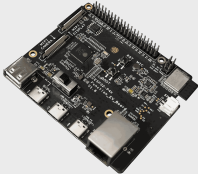
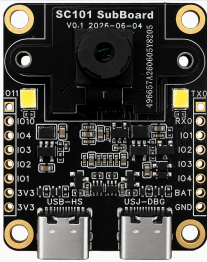
See [Get Started](#) to build and flash the firmware, and [Solution Architecture](#) for how the pieces fit together.

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-S31-Korvo-1	ESP32-S3	Supported Modules: sensor, image, display, espdl, tflite, imageio MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-EYE	ESP32-P4	Supported Modules: sensor, image, display, espdl, tflite, 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, tflite, imageio, h264, rtsp, barcode MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-VISION	ESP32-P4	Supported Modules: sensor, image, display, espdl, tflite, imageio, h264, rtsp MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	AtomS3R-M12	ESP32-S3	Supported Modules: sensor, image, display, espdl, tflite, 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.

Note: When building on Windows, install [Visual Studio Build Tools](#) before running `idf.py`. In the Visual Studio Build Tools installer, select the “Desktop development with C++” workload and make sure the “MSVC v143 - VS 2022 C++ x64/x86 build tools” component is selected.

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 `sports/esp32/boards/<BOARD>/`. `lib/micropython` is never dirtied.

3.3 Common idf.py Commands

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

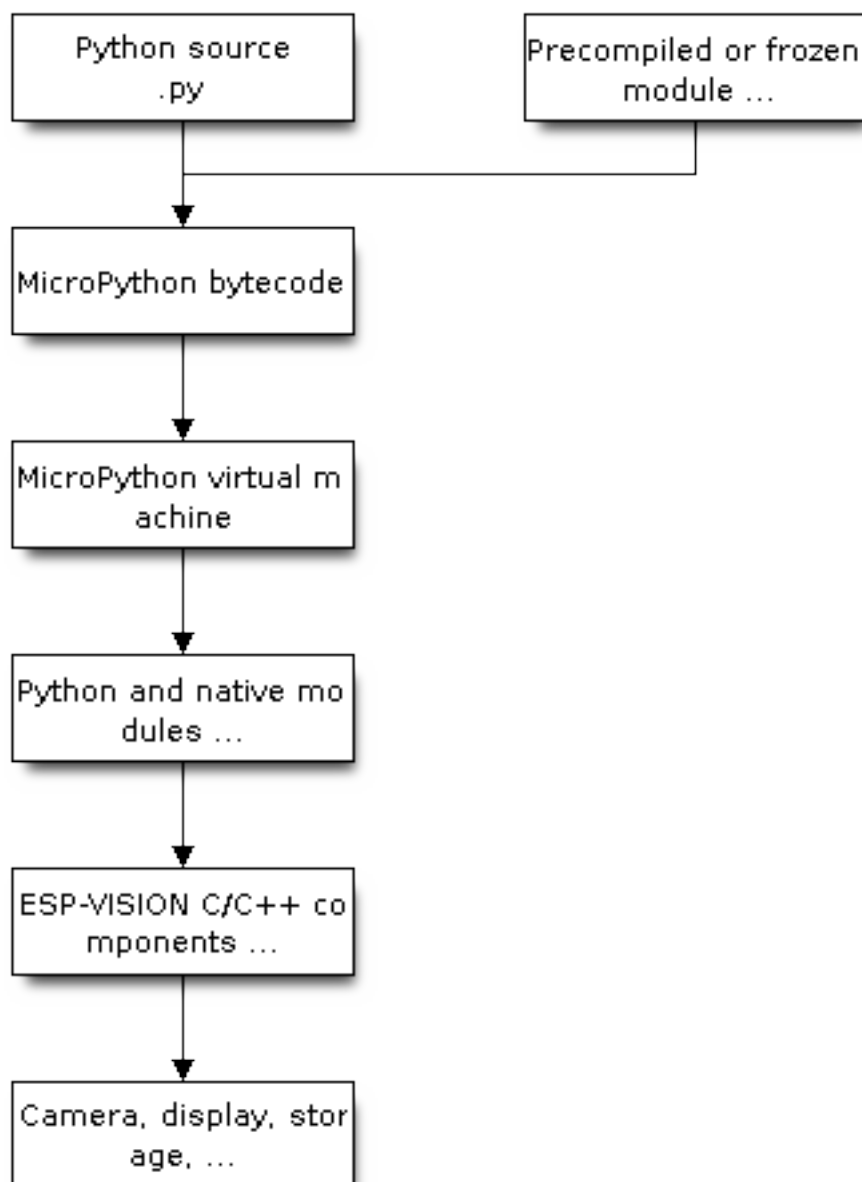
This section explains the ideas behind the ESP-VISION API: how images are represented in memory, how the image-processing algorithms work, how neural network inference runs on the device, and how frames move through the camera and codec pipelines. Read it alongside the [API Reference](#) to understand *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
<code>None</code> and <code>bool</code>	Immutable	Represent absence and state flags without defining custom sentinel objects.
<code>int</code> and <code>float</code>	Immutable	Store counters, coordinates, thresholds, and scores. Repeated arithmetic can create new objects; avoid floating-point work in hard interrupt context.
<code>str</code>	Immutable	Store text, paths, labels, and JSON keys. Concatenation creates a new string, so avoid building large strings every frame.
<code>bytes</code>	Immutable	Store compact binary constants, encoded packets, and read-only model or protocol data.
<code>bytearray</code>	Mutable	Provide reusable binary buffers for peripheral I/O and packet assembly without allocating a new object for every operation.
<code>list</code>	Mutable	Store ordered results whose size changes. <code>append()</code> and growth may allocate memory, so bound or reuse lists in sustained frame loops.
<code>tuple</code>	Immutable	Represent fixed records such as rectangles and detection results. Constant tuples can be more memory-efficient than repeatedly created lists.
<code>dict</code>	Mutable	Represent configuration and shared scalar state. Adding keys can resize the table; create the expected structure during initialization.
<code>set</code>	Mutable	Perform membership and uniqueness checks when the additional hash-table memory is justified.
<code>range</code>	Immutable	Describe integer iteration without constructing a list of all values.
<code>memoryview</code> and <code>array</code>	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 <code>machine</code> , <code>micropython</code> , <code>esp</code> , and <code>esp32</code> 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 <code>.mpy</code> , or frozen into firmware; desktop <code>pip</code> and arbitrary binary wheels must not be assumed.	Commonly uses <code>pip</code> , 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 <code>asyncio</code> , enabled <code>_thread</code> with a GIL, and scheduled GPIO IRQ and Timer callbacks; peripheral availability still depends on the selected chip and board.	Provides <code>asyncio</code> , 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 `mipremote` 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 `mipremote mip` command remains available and is the default package installation method.

```
mipremote connect <PORT> mip install --target=/lib <PACKAGE>  
mipremote connect <PORT> mip install --target=/lib <PACKAGE>@<VERSION>  
mipremote 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 `mipremote` 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:

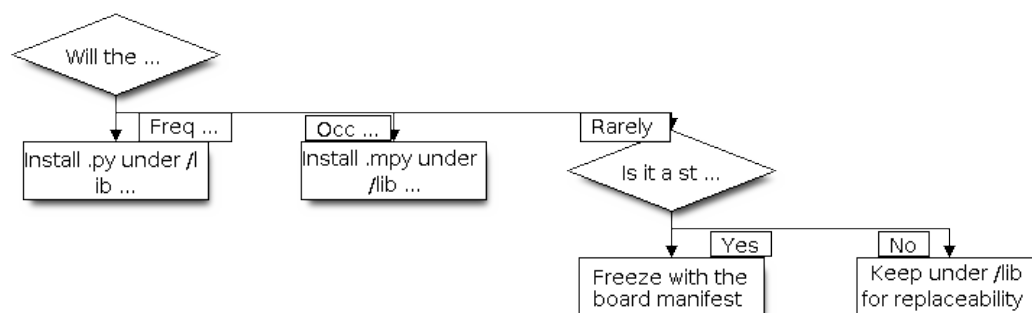


Fig. 1: Package deployment decision

Table 1: Deployment method comparison

Property	<code>.py</code> under <code>/lib</code>	<code>.mpy</code> under <code>/lib</code>	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 <code>mpremote mip</code>	Replace compatible <code>.mpy</code> 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 <code>image.Image.binary()</code> .
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 <code>image.Image.to_jpeg()</code> / <code>to_png</code> .

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)

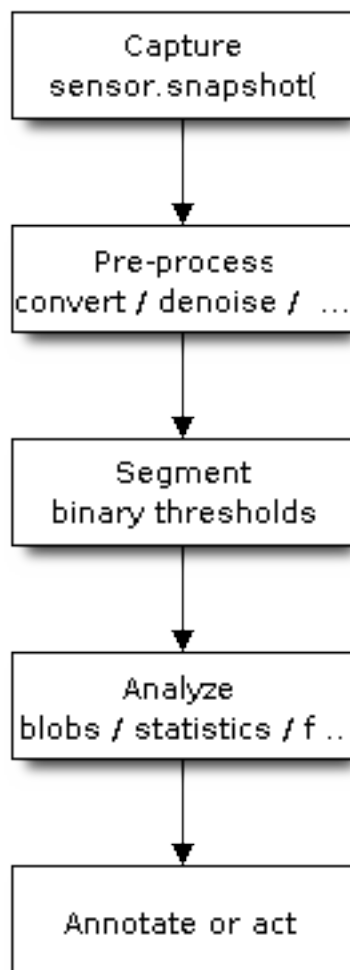
Most analysis and conversion methods accept an `roi=(x, y, w, h)` keyword that restricts the operation to a sub-rectangle of the image. Working on an ROI is both faster (fewer pixels) and more selective (ignore irrelevant areas). The ROI is always expressed in the source image's pixel coordinates.

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_line_segments()` uses Edge Drawing Lines; its `threshold` is the minimum Sobel gradient magnitude, while `merge_distance` and `max_theta_diff` control segment merging.
- `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

ESP-VISION provides on-device neural-network execution through *espdL-Model Inference*, which uses [ESP-DL](#), and *tfLite-Model Inference*, which uses TensorFlow Lite Micro for TensorFlow Lite `.tflite` models. The available APIs range from task-oriented helpers to general model execution, and model-specific preprocessing or post-processing can live in module bindings, helper code, or Python scripts depending on the model family.

4.5.1 Model Runtimes

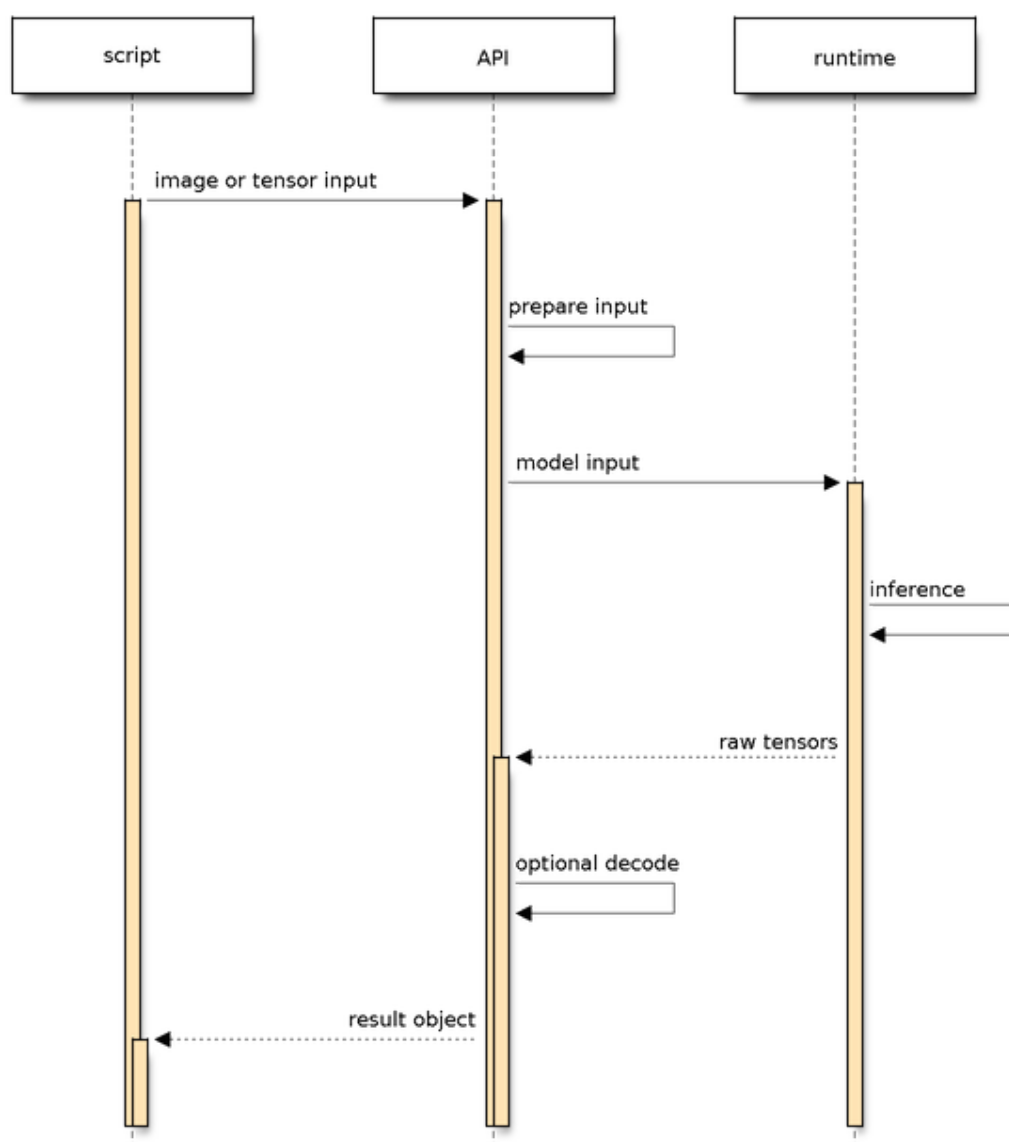
ESP-DL and TFLite Micro are the current model runtime paths exposed by ESP-VISION. The `espdL` module works with `.espdL` files, and `tfLite.Model` works with `.tflite` files. Model files are stored on board storage, such as `/sdcard` or `/flash`, and loaded at runtime. The API shape, quantization metadata, and input/output interpretation are defined by the selected runtime and model.

4.5.2 Model Files and Quantization

Models are usually exported or converted offline from a training framework before they are copied to board storage. On microcontrollers, quantized models are preferred because they reduce model size, activation memory, and arithmetic cost. Quantization maps floating-point tensor ranges onto integer values with scale and zero-point metadata; the exact metadata and file layout depend on the runtime and conversion toolchain.

4.5.3 Inference flow

A single inference call usually combines several stages: prepare input data, run the quantized network, and decode raw outputs into values the application can use:



4.5.4 Pre-processing

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

- **Shape** must match the model input tensor, including resolution and channel count for image models.
- **Color** order and pixel format must match the training pipeline.
- **Normalization or quantization** must use the scale, zero point, mean, standard deviation, or other transform expected by the model. Some APIs expose these as parameters, while examples may show them directly in Python.

4.5.5 Post-processing

Raw network outputs often need task-specific decoding before they are useful to an application:

- **Object detection** ([`espdl.PedestrianDetect`](#), [`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.

`PedestrianDetect` and `YOLO11` also cap 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.
- **Raw output decoding** (`espdl.Model`, `tflite.Model`) exposes model outputs before task decoding. Application code must unpack the tensor dtype, use the quantization metadata, and run model-specific steps such as anchor decode, sigmoid or softmax, NMS, top-k selection, and coordinate mapping.

Thresholds and result limits can often be tuned at runtime without reloading the model, which is useful when adapting to lighting, distance, or scene density.

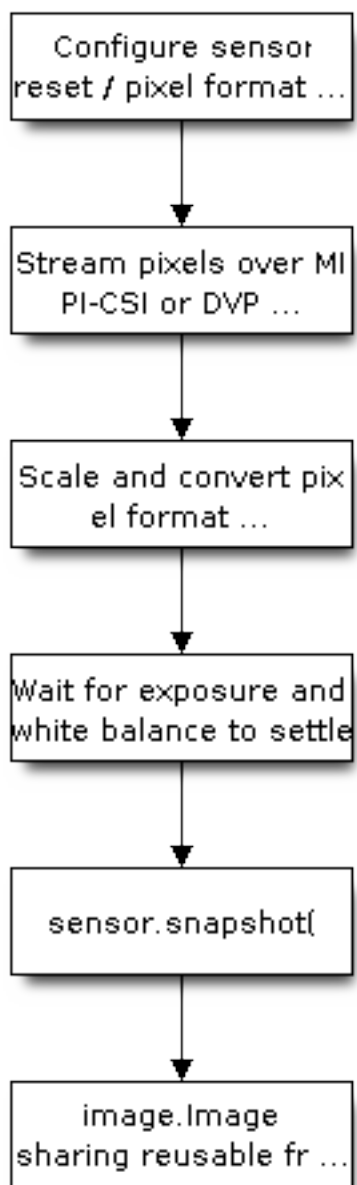
4.5.6 Memory and performance

Models and their activation buffers are large and are allocated in PSRAM. Load a model once and reuse it across frames rather than recreating it per frame. Call `deinit()` (or drop the last reference) to free a model when you are done. Inference cost usually scales with input resolution and model size, so smaller models generally 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

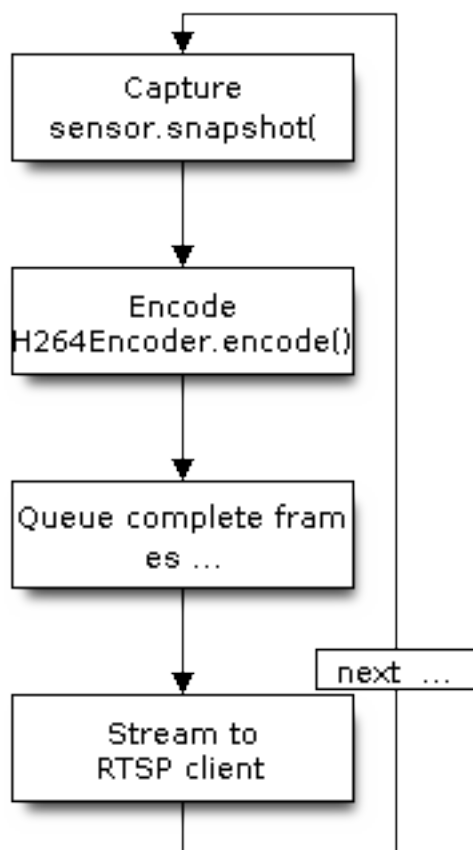
For continuous video, per-frame JPEG is wasteful because it cannot exploit the similarity between consecutive frames. `h264.H264Encoder` produces an H.264 stream where occasional **keyframes** (IDR/I) are self-contained and the frames between them only encode what changed. The encoder's parameters tune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

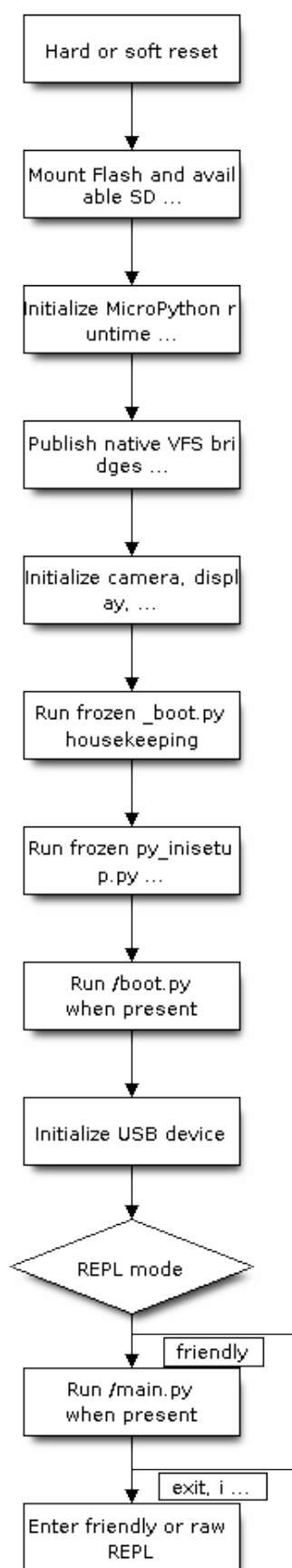
A hard reset restarts the MCU and ESP-IDF runtime before creating a new MicroPython environment. It occurs after power-on, the board reset button, `machine.reset()`, watchdog or brownout reset, and wake-up from deep sleep. Use `machine.reset_cause()` when an application needs to distinguish the reset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



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 ESP-IDF storage manager owns the board filesystems independently of the MicroPython VM. It mounts the `ffat` partition internally at `/flash` and mounts an available SD card at `/sdcard`. On each VM start, native MicroPython VFS bridges publish the Flash volume at `/` and the SD volume at `/sdcard`. A soft reset recreates only these bridge objects and does not remount an already mounted physical filesystem; if the SD card was not mounted during cold boot, the mount is retried on each subsequent VM start so that a card inserted later becomes available after a soft reset.

An unformatted `ffat` partition is formatted as FAT during storage initialization. A legacy partition named `vfs` is only mounted when it already contains FAT and is never automatically formatted, so a possible LittleFS2 volume is not silently erased. Other mount or write failures are reported without destructive repair. The frozen `py_inisetup.py` then creates `/boot.py`, `/main.py`, `/README.txt`, and the `/.esp_vision_disk` marker when they do not already exist.

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`, `espd1`, and `tf1ite`.

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`, `espd1`, 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"))

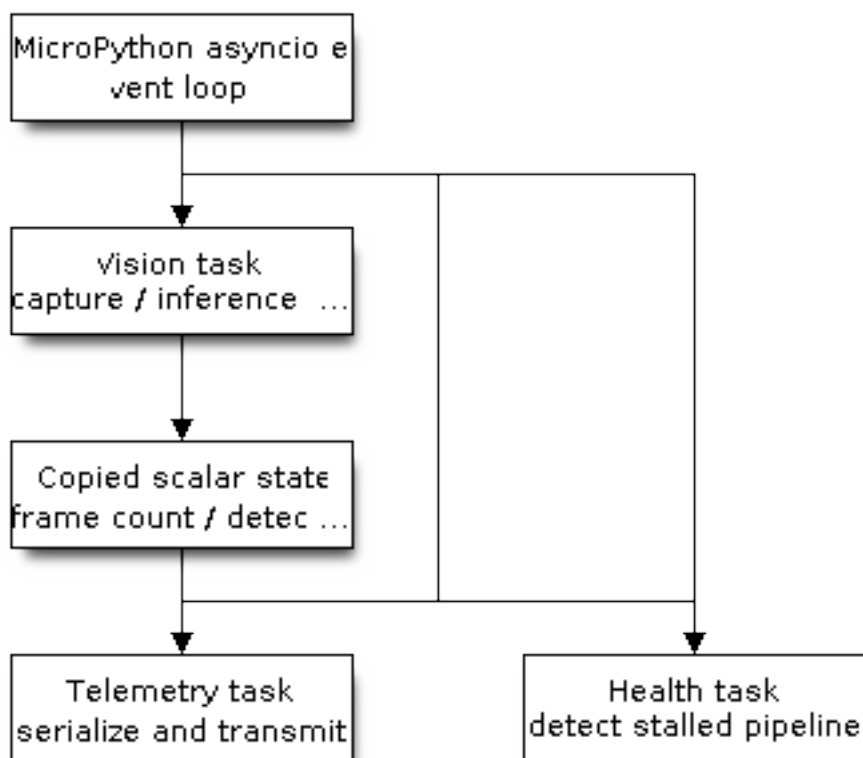
gc.collect()
print("free heap:", gc.mem_free())
```

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 espd1
import json
import sensor
import time

MODEL = "/sdcard/espdet_pico_224_224_face.espd1"

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`

`sensor.VGA`

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(pixel_format)`

Select the output pixel format.

Parameters

pixel_format –sensor.GRAYSCALE or sensor.RGB565.

`sensor.get_pixformat()`

Return the current pixel format constant.

`sensor.set_framesize(frame_size)`

Select the output frame size.

Parameters

frame_size –sensor.QQVGA, sensor.QVGA, or sensor.VGA when supported by the board.

`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, line segments, 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, Segment, 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 segment in img.find_line_segments(threshold=50, merge_distance=10):
    img.draw_line(segment.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)
```

`find_lines()` and `find_circles()` use Hough accumulator thresholds. `find_line_segments()` uses the Edge Drawing Lines detector, where `threshold` is the minimum Sobel gradient magnitude and `merge_distance` joins nearby segments. Increasing thresholds 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, line segments, 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).

cx_f()
Centroid x (float).

cy()
Centroid y (integer).

cy_f()
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/stddev/...) 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 as a binary EV-MUX `preview.frame JPEG`.

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/TRANPOSE/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_line_segments (**, roi=..., merge_distance=..., max_theta_diff=..., threshold=...*)

Find line segments with the Edge Drawing Lines detector. `merge_distance` and `max_theta_diff` merge nearby, similarly angled segments.

Parameters

threshold –minimum Sobel gradient magnitude used to seed and trace edges.

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 (`PedestrianDetect`, `ESPDet`, `YOLO11`), pose estimation (`YOLO11nPose`), and image classification (`ImageNetCls`), plus `Model` for exposing raw ESP-DL output tensors to Python.

5.5.1 Raw Output Tensors

```
import sensor, espdl

model = espdl.Model("/sdcard/custom.espdl", mean=(0, 0, 0), std=(255, 255, 255),
↳ letterbox=True)
try:
    print("inputs:", model.inputs())
    print("outputs:", model.outputs())
    img = sensor.snapshot()
    outputs = model.predict(img)
    for name, tensor in outputs.items():
        _, shape, dtype, exponent, raw = tensor
        print(name, shape, dtype, exponent, len(raw))
finally:
    model.deinit()
```

Use `Model` when the built-in task wrappers do not match a model's output layout. ESP-DL still performs image preprocessing and inference; `predict()` returns the raw output tensors so Python code can run the model-specific decode. `inputs()` and `outputs()` return `TensorInfo` tuples (name, shape, dtype, exponent). `predict()` returns `RawTensor` tuples (name, shape, dtype, exponent, bytes) keyed by output tensor name. Decode the raw bytes according to dtype, apply the ESP-DL power-of-two exponent scale, and run logic such as sigmoid, box decode, NMS, softmax, or top-k selection. When `letterbox=True`, remove the model-input padding before mapping coordinates back to the source image.

5.5.2 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.

5.5.3 PicoDet

`PicoDet` is the detector family used by the bundled pedestrian model. Its anchor-point output tensors are decoded by ESP-DL's `Pico` postprocessor; this output layout is distinct from `ESPDet` and `YOLO11`.

5.5.4 PedestrianDetect

PedestrianDetect is the Python API for the bundled 224x224 PicoDet pedestrian model. It uses the model's preprocessing defaults and returns the standard detection tuples:

```
det = espdl.PedestrianDetect(
    "/sdcard/pedestrian_detect/pedestrian_detect_pico.espd1",
    score=0.7,
    nms=0.5,
)
```

5.5.5 Image Classification

```
import espdl, image

img = image.Image("/sdcard/cat.jpg").to_rgb565(copy=True)
classifier = espdl.ImageNetCls(
    "/sdcard/imagenet_cls_mobilenetv2_s8_v1.espd1",
    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.

5.5.6 Pose Estimation

```
pose = espdl.YOLO11nPose("/sdcard/espdet_yolo11n_pose_160_160_coco.espd1", 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.

5.5.7 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.

5.5.8 Result tuples

- `TensorInfo`: (name, shape, dtype, exponent)
- `RawTensor`: (name, shape, dtype, exponent, bytes)
- `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 `.espd` 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` (pedestrian detection, ESP-DL raw output decoding, ESPDet, YOLO11, pose, ImageNet classification).

5.5.9 Functions

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

Preload an ESP-DL model file.

Parameters

- **path** –model path, for example “/sdcard/model.espd” or “/flash/model.espd” .
- **profile** –True enables ESP-DL profiling output when supported.

5.5.10 Classes

class `espd.Model(path, *, mean=..., std=..., rgb_swap=..., letterbox=..., pad=...)`

Generic ESP-DL model runner for examples that implement model-specific decoding in Python.

Create a raw model runner from an `.espd` model.

Parameters

- **path** –model path.
- **mean** –optional RGB mean values for image preprocessing.
- **std** –optional RGB standard deviation values for image preprocessing.
- **rgb_swap** –True converts RGB input to BGR before model inference.
- **letterbox** –True keeps aspect ratio and pads the model input.
- **pad** –optional RGB padding values used when letterbox is enabled.

deinit ()

Release model resources.

inputs ()

Return model input metadata keyed by tensor name.

outputs ()

Return model output metadata keyed by tensor name.

predict (image)

Run model inference on an image and return raw output tensors keyed by tensor name.

Parameters

- **image** –RGB565 or grayscale image.

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

ESP-DL object detection wrapper for ESPDet models.

Create a detector from an `.espd` 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.PedestrianDetect` (*path*, *, *score*=..., *nms*=..., *topk*=...)

ESP-DL PicoDet pedestrian detection wrapper.

Create a pedestrian detector from the bundled ESP-DL model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold; defaults to the upstream model's 0.7.
- **nms** –optional non-maximum suppression threshold; defaults to the upstream model's 0.5.
- **topk** –maximum number of detections returned per frame.

deinit ()

Release model resources.

detect (*image*)

Run pedestrian detection on an 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 `espd1.YOLO11nPose` (*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11n COCO pose wrapper.

Create a YOLO11n pose detector from an .espd1 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 `espd1.ImageNetCls` (*path*, *, *topk*=..., *score*=..., *mean*=..., *std*=..., *softmax*=...)

ESP-DL ImageNet classification wrapper.

Create a classifier from an .espd1 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 tflite –Model Inference

The `tflite` module loads TensorFlow Lite `.tflite` flatbuffers and runs them through TensorFlow Lite Micro. It provides a general model execution interface for TFLite models, while higher-level task helpers can be layered on top when a model family needs shared preprocessing or post-processing.

5.6.1 Basic Usage

```
import tflite

model = tflite.Model("/sdcard/sine.tflite")
try:
    print("input:", model.input_shape, model.input_dtype, model.input_scale, model.
    ↪input_zero_point)
    print("output:", model.output_shape, model.output_dtype, model.output_scale, ↪
    ↪model.output_zero_point)
    outputs = model.predict([input_array])
    raw = outputs[0]
finally:
    model.deinit()
```

`predict()` returns raw output tensors as `ulab` ndarrays. Quantized outputs are not automatically decoded; use `output_scale` and `output_zero_point` to convert values when the model requires it.

5.6.2 Callable Inputs

For image models it is often cheaper to fill the input tensor directly instead of building an intermediate ndarray:

```
def fill_input(buffer, shape, dtype_code):
    # Fill buffer with model-specific quantized bytes.
    ...

outputs = model.predict([fill_input])
```

The callable receives a writable bytearray view of the raw input tensor, the tensor shape, and an integer dtype code such as `ord("b")` for `int8`. This path is useful for image resizing, grayscale conversion, and quantization in Python examples.

5.6.3 Post-processing

`Model` can take a default `postprocess` callback, and `predict()` can take a one-shot callback. The callback receives `(model, inputs, raw_outputs)` and can return any Python object.

See also:

Runnable examples: `example/03-Machine-Learning/01-TFLite` (person/no-person and sine smoke tests).

5.6.4 Classes

class `tf.lite.Model` (*path*, *, *postprocess*=...)

TensorFlow Lite Micro model wrapper.

Load a .tflite flatbuffer from the filesystem.

Parameters

- **path** –model path, for example “/sdcard/model.tflite” or “/flash/model.tflite” .
- **postprocess** –optional callback that receives (model, inputs, raw_outputs) and returns the final result.

deinit ()

Release model, interpreter, and tensor arena resources.

predict (*inputs*, *, *callback*=...)

Run inference.

Parameters

- **inputs** –one item per input tensor. Each item is an ulab ndarray or a callable that fills the raw tensor buffer.
- **callback** –optional one-shot post-processing callback; overrides the default postprocess callback.

5.7 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`.

5.7.1 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.

5.7.2 Replay a Recorded Stream

```
import image

stream = image.ImageIO("/sdcard/stream.bin", "r")
try:
    while True:
        img = stream.read(loop=False, pause=True)
        if img is None:
            break
        img.flush()
```

(continues on next page)

(continued from previous page)

```
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.

5.7.3 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.

5.7.4 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.

5.7.5 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.

5.8 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.

5.8.1 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.

5.8.2 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.

5.8.3 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`.

5.8.4 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.

5.9 rtsp –RTSP Streaming

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

5.9.1 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:
    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.

5.9.2 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.

5.9.3 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`.

5.9.4 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`.

```
/*
 * 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 `user_mod_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 supports two model runtime paths: ESP-DL `.espd` models use the [espd –Model Inference](#) module, and TensorFlow Lite `.tflite` models use the [tflite –Model Inference](#) module. Model files 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

Choose the model runtime first:

- ESP-DL: get a ready `.espd` from the [ESP-DL model zoo](#), or convert your own model to the `.espd` format with the ESP-DL quantization/export toolchain, matching the selected chip (ESP32-P4, ESP32-S3, or ESP32-S31).
- TFLite Micro: use a TensorFlow Lite `.tflite` flatbuffer that fits TensorFlow Lite Micro and the enabled operator set. Quantized int8 models are usually the practical target on the current boards.

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

6.3.2 2. Copy the Model to Board Storage

Place the `.espd` or `.tflite` 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 API

Choose the API that matches the runtime and task:

Task	API	Result
ESP-DL object detection (ESPDet)	<code>espd.ESPDet</code>	<code>(x, y, w, h, score, category)</code>
ESP-DL object detection (YOLO11)	<code>espd.YOLO11</code>	<code>(x, y, w, h, score, category)</code>
ESP-DL pose detection	<code>espd.YOLO11nPose</code>	detection plus 17 COCO keypoints
ESP-DL image classification	<code>espd.ImageNetCls</code>	<code>(label, score)</code>
ESP-DL with Python decoding	<code>espd.Model</code>	raw output bytes with tensor metadata
Generic TFLite Micro execution	<code>tflite.Model</code>	raw output tensors, or a callback result

For ESP-DL wrappers, pass `mean`, `std`, `rgb_swap`, `score`, `nms`, `topk`, or `softmax` to the constructor when the model needs different preprocessing or filtering. When you need to keep ESP-DL inference but decode outputs in Python, inspect `espd.Model.inputs()` and `espd.Model.outputs()`, then decode the `RawTensor` bytes returned by `predict()`. For TFLite Micro models, inspect `input_shape`, `input_dtype`, `input_scale`, `input_zero_point`, `output_shape`, `output_dtype`, `output_scale`, and `output_zero_point` and implement the model-specific preprocessing or post-processing in Python or helper code.

6.3.4 4. Run ESP-DL 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()
```

6.3.5 5. Decode ESP-DL Outputs in Python

Use `espdl.Model` when the model can use ESP-DL image preprocessing and inference, but its output tensors need Python-side decoding.

```
import sensor, espdl

model = espdl.Model("/sdcard/my_model.espdl", mean=(0, 0, 0), std=(255, 255, 255),
↳ letterbox=True)
try:
    print("inputs:", model.inputs())
    print("outputs:", model.outputs())
    img = sensor.snapshot()
    outputs = model.predict(img)
    for name, tensor in outputs.items():
        _, shape, dtype, exponent, raw = tensor
        print(name, shape, dtype, exponent, len(raw))
finally:
    model.deinit()
```

Each output tensor is returned as raw bytes plus shape, dtype, and ESP-DL exponent metadata. Decode the bytes according to the tensor type, apply the exponent scale, and then run model-specific post-processing such as sigmoid, box decode, NMS, classification top-k, or coordinate unletterboxing.

6.3.6 6. Run TFLite Micro Inference

```
import tflite

def fill_input(buffer, shape, dtype_code):
    # Fill model-specific quantized input bytes.
    ...

model = tflite.Model("/sdcard/my_model.tflite")
try:
    print("input:", model.input_shape, model.input_dtype, model.input_scale, model.
↳ input_zero_point)
    print("output:", model.output_shape, model.output_dtype, model.output_scale,
↳ model.output_zero_point)
    outputs = model.predict([fill_input])
    raw = outputs[0]
finally:
    model.deinit()
```

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

03-Machine-Learning/01-TFLite/ for TFLite Micro scripts (`person_detection.py` and `sine.py`).

6.3.7 7. Optional: Profiling and Validation

Use `espdml.load_model()` with `profile=True` to emit ESP-DL profiling output when verifying a new .espdml model's performance. For TFLite Micro models, print `model.len`, `model.ram`, input metadata, and output metadata to verify flash size, arena size, tensor layout, and quantization before tuning post-processing.

6.4 Train an ESPDet Pico Model

ESP-VISION runs ESPDet Pico object detection models through `espdml.ESPDet`. Model training, export, and quantization are handled by the ESP-Detection project. ESP-VISION loads the exported .espdml file and runs real-time inference through the `sensor`, `image`, and `espdml` modules.

This guide uses a single-class object detection model as an example and covers the complete workflow from environment setup, dataset preparation, training, quantized export, and running the model on ESP-VISION.

6.4.1 1. Prepare the ESP-Detection Environment

Use the following training repository:

```
git clone -b feat/esp-vision-train https://github.com/YanKE01/esp-detection.git
cd esp-detection
```

ESP-Detection uses a Python training environment. ESP-IDF is not required for training. ESP-IDF is only required when building ESP-VISION firmware or running a C++ example project generated for ESP-DL.

ESP-Detection uses `uv` to manage dependencies. Select the installation command for the operating system.

Linux/macOS:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows PowerShell:

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

After installation, reopen the terminal or refresh the current terminal's `PATH` as instructed by the installer. After confirming that `uv` is available, synchronize the project environment:

```
uv sync
```

`uv sync` installs Ultralytics, PyTorch, ONNX, ONNX Runtime, OpenCV, and `esp-ppq`. Training and quantization are much faster on machines with an NVIDIA GPU. CPU execution is also supported, but training time is significantly longer.

After the environment is ready, check that the command entry points are available:

```
uv run python espdet_train.py --help
uv run python -m deploy.quantize_aligned --help
```

6.4.2 2. Prepare the Dataset

When preparing a dataset with Roboflow, export it in YOLOv11 format. After export, organize the dataset in the ESP-Detection repository, for example:

```
datasets/my_object/
  images/
    train/
    val/
  labels/
    train/
    val/
```

ESP-Detection reads the YOLO detection label format. Each image has a same-named `.txt` label file under `labels`. For example, `images/train/0001.jpg` corresponds to `labels/train/0001.txt`. Each line in a label file describes one bounding box:

```
class_id x_center y_center width height
```

`x_center`, `y_center`, `width`, and `height` are floating-point values normalized by image width and height, usually in the 0.0 to 1.0 range.

For example, a single-class safety helmet dataset can use:

```
0 0.5123 0.4381 0.1840 0.1365
```

Dataset labels should match the class names shown later on the ESP-VISION side.

Then create a dataset configuration under `cfg/datasets/`, for example `cfg/datasets/my_object.yaml`:

```
path: datasets/my_object
train: images/train
val: images/val
test:

names:
  0: my_object
```

`path` is the dataset root directory. `train` and `val` are image directories relative to `path`. The order of `names` is important because it defines the meaning of the `category` value returned by `espdet.ESPDet.detect()` on the board. Keep this order stable after training.

For a multi-class model, add more classes:

```
names:
  0: helmet
  1: person
  2: vest
```

6.4.3 3. Train ESPDet Pico

ESP-Detection provides `espdet_train.py` as the train-only entry point. It builds the ESPDet Pico network from `cfg/models/espdet_pico.yaml` and prints the best checkpoint path.

Single-class training example:

```
uv run python espdet_train.py \
  --class_name my_object \
  --pretrained_path None \
  --dataset cfg/datasets/my_object.yaml \
  --size 320 320
```

Key parameters:

Parameter	Description
<code>--class_name</code>	Training task name. It affects the Ultralytics output directory name.
<code>--pretrained_path</code>	Pretrained checkpoint path. Use <code>None</code> when no pretrained checkpoint is available; the script builds a new model from <code>cfg/models/espdet_pico.yaml</code> .
<code>--dataset</code>	Dataset YAML path.
<code>--size</code>	Input size in height width format. For example, <code>320 320</code> means 320x320 input.

The training script uses long training by default, with a large batch size, RAM cache, and data augmentation. For small datasets, monitor validation curves to avoid overfitting. When training finishes, the terminal prints output similar to:

```
Training complete: runs/detect/my_object
Best weights: runs/detect/my_object/weights/best.pt
```

Use `weights/best.pt` for quantization.

6.4.4 4. Prepare Calibration Data

Exporting `.espd1` requires int8 quantization, and quantization requires calibration images. Calibration images should cover the real deployment distribution, including different lighting conditions, distances, angles, backgrounds, and object sizes.

The simplest approach is to use validation images directly:

```
datasets/my_object/images/val
```

A separate calibration directory can also be prepared:

```
deploy/my_object_calib/
0001.jpg
0002.jpg
0003.jpg
```

Calibration images do not need labels, but their distribution should be close to actual camera input. If the calibration set is too narrow, simulator metrics may look normal while on-board scores are low or detections are missing.

6.4.5 5. Quantize and Export `.espd1`

Use `deploy.quantize_aligned` to export the trained `best.pt` checkpoint to an ESP-DL `.espd1` model:

```
uv run python -m deploy.quantize_aligned \
  --model runs/detect/my_object/weights/best.pt \
  --size 320 320 \
  --target esp32p4 \
  --calib_data datasets/my_object/images/val \
  --espd1 deploy/espdet_pico_320_320_my_object.espd1 \
  --data cfg/datasets/my_object.yaml
```

Parameters:

Parameter	Description
<code>--model</code>	Trained best <code>.pt</code> checkpoint. An exported ONNX model can also be used.
<code>--size</code>	Model input size. It must match the training size.
<code>--target</code>	Target chip, such as <code>esp32p4</code> or <code>esp32s3</code> . It should match the chip that will run the model.
<code>--calib_data</code>	Calibration image directory.
<code>--espdl</code>	Output ESP-DL model path.
<code>--data</code>	Dataset YAML. When provided, the script also reports simulator validation metrics.

The quantization script prepares ONNX from `pt` internally, then exports an int8 ESP-DL model through ESP-PPQ. The current configuration uses percentile calibration, bias correction, and fusion settings tuned for scale consistency. The script prints the exported model's operator histogram. When `--data` is provided, it also prints simulator `mAP50` and `mAP50-95`.

After quantization, confirm that the `.espdl` file exists:

```
ls -lh deploy/espdet_pico_320_320_my_object.espdl
```

6.4.6 6. Add Model Metadata

To publish the model as a shared ESP-VISION asset, place it under `models/espdet/pico/<name>/:`

```
models/espdet/pico/my_object/
  README.md
  espdet_pico_320_320_my_object.espdl
  espdet_pico_320_320_my_object.json
```

The `.espdl` file is the model binary. The `.json` file is sidecar metadata read by model lists and tooling. Minimal JSON example:

```
{
  "name": "ESPDet Pico My Object",
  "architecture": "ESPDet Pico",
  "api": "espdl.ESPDet",
  "task": "detection",
  "input": "320x320",
  "inputFormat": "RGB565",
  "dataset": "My Object",
  "labels": ["my_object"],
  "sizeBytes": 574864,
  "description": "Detects my_object in camera images."
}
```

`labels` must follow the same order as names in the dataset YAML. `sizeBytes` should be the actual model file size. On Linux, use:

```
stat -c '%s' models/espdet/pico/my_object/espdet_pico_320_320_my_object.espdl
```

`README.md` should describe the model purpose, dataset source, input size, quantization metrics, and basic usage. For public datasets, keep the dataset link in README instead of the JSON file.

6.4.7 7. Copy the Model to Board Storage

ESP-VISION loads models from the file system at runtime. Two common paths are:

Path	Use
/sdcard	SD card. Suitable for multiple models, large files, or frequent model replacement.
/flash	On-flash FAT data partition. It is usually exposed as a USB MSC drive and is suitable for a small number of frequently used models.

For example, after copying the model to the SD card root directory, use:

```
MODEL = "/sdcard/espdet_pico_320_320_my_object.espd1"
```

If the model is copied to the flash root directory, use:

```
MODEL = "/flash/espdet_pico_320_320_my_object.espd1"
```

6.4.8 8. Run on ESP-VISION

The following is a complete MicroPython inference script:

```
import espd1
import sensor
import time

MODEL = "/sdcard/espdet_pico_320_320_my_object.espd1"
LABELS = ("my_object",)

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

det = espd1.ESPDet(MODEL, score=0.5, nms=0.7)

try:
    while True:
        img = sensor.snapshot()
        for x, y, w, h, score, category in det.detect(img):
            label = LABELS[category] if category < len(LABELS) else str(category)
            img.draw_rectangle(x, y, w, h, color=(255, 0, 0), thickness=2)
            img.draw_string(x, max(0, y - 12), "%s %.2f" % (label, score),
↪color=(255, 0, 0))
            img.flush()
            time.sleep_ms(20)
finally:
    det.deinit()
```

sensor output should use RGB565. score is the confidence threshold, and nms is the NMS threshold. During the first on-board test, start with a lower score to check whether the model can produce stable boxes, then adjust it based on false positives and missed detections.

6.4.9 9. Validate and Tune

Use the following validation order:

- Check the training log and validation metrics first to confirm that the floating-point model does not obviously underfit or overfit.
- Check the simulator metrics reported by the quantization script to confirm that the int8 model does not drop significantly.

- Run the model with real camera input on ESP-VISION, and inspect detections at different distances, angles, and lighting conditions.
- Tune `score` and `nms`, then freeze the final model, labels, and example script.

If simulator metrics are normal but on-board results are poor, check these items first:

- Whether `--size` exactly matches the training size.
- Whether `labels` follows the same order as `names` in the dataset YAML.
- Whether calibration images cover the real camera input distribution.
- Whether the board camera image has exposure issues, color issues, or objects that are too small.
- Whether `score` is too high and filters out low-score detections.

After the model file, JSON metadata, README, and test script are verified, the model can be added to ESP-VISION's `models/` and `example/` directories as a reusable model.

6.5 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.

6.5.1 MicroPython Port Side

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

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

6.5.2 ESP-VISION Side

In `boards/<BOARD>/`:

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

6.5.3 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.

6.6 Agent Code Generation and EV-MUX Closed-Loop Test

This guide is normative for AI agents that generate ESP-VISION application code. Treat the target as an ESP32 microcontroller running the ESP-VISION build of MicroPython v1.28.0, not as a PC running CPython. Use only APIs present in this knowledge pack, enabled for the selected board, or demonstrated by a repository example; never assume that a desktop standard-library module, `pip` package, native wheel, operating-system service, subprocess, or PC filesystem is available on the device.

6.6.1 Dependency Rules

The default ESP-VISION firmware cannot dynamically install dependencies from `requirements.txt` and does not include device-side `mip`. Generated target code must not invoke `pip`, parse `requirements.txt`, call `mip.install()`, or silently depend on an unverified third-party package. If a required module is not in the knowledge pack, either rewrite the solution with supported modules or explicitly state that it must be copied to `/lib` with a host tool, precompiled as compatible `.mpy`, frozen into custom firmware, or implemented as a native module before the script can run.

Host-side tooling is separate from target code. A PC utility may use CPython packages such as `pyserial`, but label that code as host-only and never import those packages from the ESP-VISION script.

6.6.2 All-in-One Output

Unless the user asks for a package or a multi-file architecture, return one self-contained `main.py` for the device. Put supported imports, constants, initialization, application logic, the main loop or bounded test, error handling, and cleanup in that file. Avoid helper modules that the user would have to create separately. Models and other binary assets may remain separate files, but list their exact device paths and deployment prerequisites.

Prefer bounded, observable verification code before an infinite product loop. Print a unique success marker only after the camera, model, storage, network, or other requested operation has really succeeded. The following target-side smoke test is an all-in-one MicroPython script and prints `EVTEST:OK` only after capturing a frame:

```
import gc
import sensor

def main():
    print("EVTEST:START")
    sensor.reset()
    try:
        sensor.set_pixformat(sensor.RGB565)
        sensor.set_framesize(sensor.QQVGA)
        sensor.skip_frames(time=500)
        image = sensor.snapshot()
        width = image.width()
        height = image.height()
```

(continues on next page)

(continued from previous page)

```
finally:
    sensor.shutdown()
print("EVTEST:OK {}x{} heap={}".format(
    width, height, gc.mem_free()))

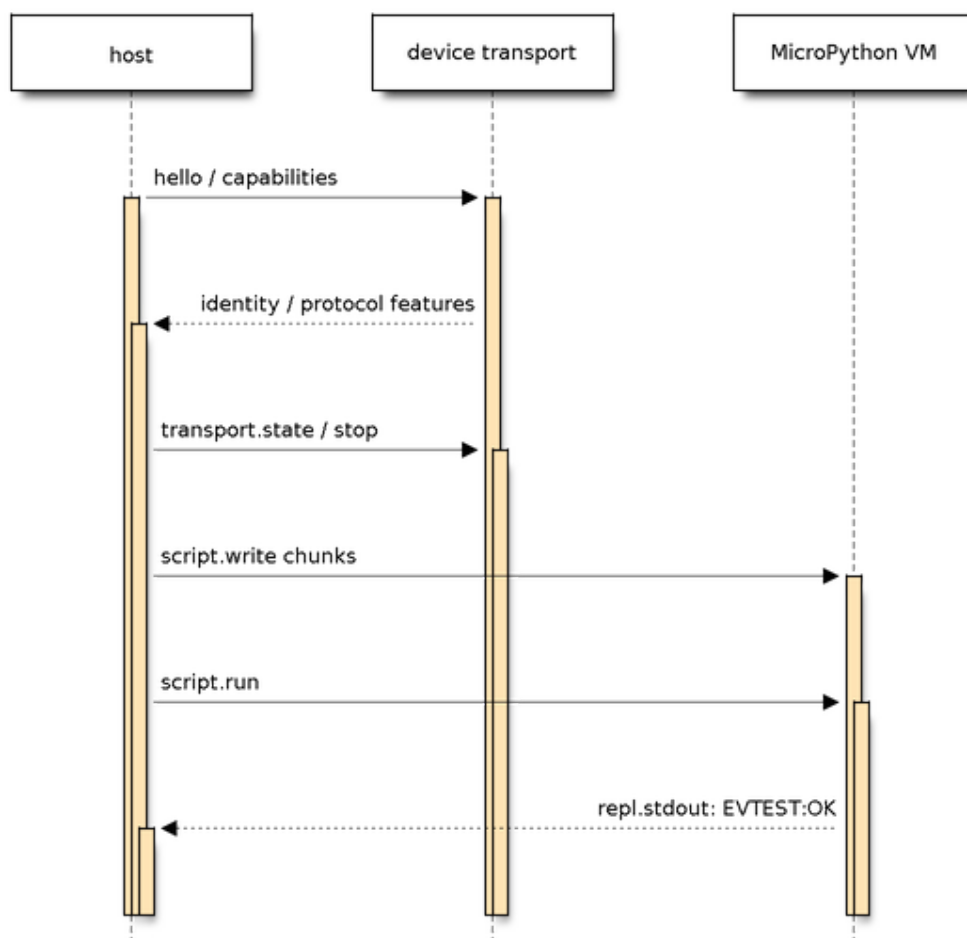
main()
```

6.6.3 USB-OTG CDC and USJ Transport Contract

ESP-VISION carries the same EV-MUX protocol over USB-OTG CDC (sink `cdc`) and USB-Serial-JTAG (sink `usj`). Open exactly one device port for a test. Opening the CDC port with DTR asserted makes CDC the automatic route; otherwise supported boards fall back to USJ. Do not open both ports or leave another serial monitor attached while testing. USB baud rate is not part of the protocol, although host serial APIs still require a placeholder baud value.

Every frame is `0x1e` followed by `EVMUX/1` `h=<metadata_len>` `p=<payload_len>` `c=<crc32>` `\r\n`, exactly `h` bytes of UTF-8 JSON metadata, exactly `p` payload bytes, and a final `0x1f`. Lengths, not delimiters, define the metadata and payload boundaries. Device frames carry the payload CRC32; a host command may send a zero CRC, although computing it is recommended. A receiver must resynchronize on `0x1e`, validate lengths and CRC, then dispatch every frame by `metadata.channel` rather than by USB port or arrival order.

The minimum closed loop is: open the selected port; exchange `hello` and `capabilities`; verify `firmware.id == "esp-vision"` and the negotiated `evMuxVersion`; query `transport.state`; stop the current script to reach the REPL; upload one complete device script with `script.write`; start it with `script.run`; and require an expected marker on `repl.stdout`. A `script.run` response only means the script was queued, so it is not a successful application test by itself.



The following host-only CPython script implements that complete loop for either USB-OTG CDC or USJ. Install `pyserial` on the PC with `python -m pip install pyserial`, save this as one file, close other serial tools, and run it with the selected device port, for example `python evmux_smoke.py /dev/ttyACM0`. The uploaded MicroPython program is embedded in the host script, so no second source file is required.

```
#!/usr/bin/env python3
import argparse
import base64
import json
import re
import sys
import time
import zlib

import serial

DEVICE_SCRIPT = """\
import gc
import sensor

def main():
    print("EVTEST:START")
    sensor.reset()
    try:
        sensor.set_pixformat(sensor.RGB565)
```

(continues on next page)

(continued from previous page)

```

        sensor.set_framesize(sensor.QQVGA)
        sensor.skip_frames(time=500)
        image = sensor.snapshot()
        width = image.width()
        height = image.height()
    finally:
        sensor.shutdown()
    print("EVTEST:OK {}x{} heap={}".format(
        width, height, gc.mem_free()))

main()
"""

SOF = b"\x1e"
EOF = b"\x1f"
HEADER_RE = re.compile(
    rb"EVMUX/1 h=([0-9]+) p=([0-9]+) c=([0-9A-Fa-f]{8})\r\n")
MAX_METADATA = 64 * 1024
MAX_PAYLOAD = 16 * 1024 * 1024
WRITE_CHUNK = 768

class EvMux:
    def __init__(self, port):
        self.port = port
        self.next_seq = 1
        self.text_tail = bytearray()

    def _read_exact(self, size, deadline):
        data = bytearray()
        while len(data) < size:
            if time.monotonic() >= deadline:
                raise TimeoutError("timed out while reading EV-MUX frame")
            part = self.port.read(size - len(data))
            if part:
                data.extend(part)
        return bytes(data)

    def read_frame(self, deadline):
        while time.monotonic() < deadline:
            if self.port.read(1) != SOF:
                continue

            header = bytearray()
            while len(header) < 80:
                char = self._read_exact(1, deadline)
                header.extend(char)
                if char == b"\n":
                    break
            match = HEADER_RE.fullmatch(bytes(header))
            if match is None:
                continue

            metadata_len = int(match.group(1))
            payload_len = int(match.group(2))
            expected_crc = int(match.group(3), 16)
            if not 0 < metadata_len <= MAX_METADATA:
                raise RuntimeError("invalid metadata length: {}".format(metadata_
↵len))

            if not 0 <= payload_len <= MAX_PAYLOAD:
                raise RuntimeError("invalid payload length: {}".format(payload_

```

(continues on next page)

(continued from previous page)

```

    len))

    metadata_bytes = self._read_exact(metadata_len, deadline)
    payload = self._read_exact(payload_len, deadline)
    if self._read_exact(1, deadline) != EOF:
        raise RuntimeError("invalid EV-MUX EOF")
    actual_crc = zlib.crc32(payload) & 0xFFFFFFFF
    if expected_crc and actual_crc != expected_crc:
        raise RuntimeError("EV-MUX payload CRC mismatch")
    metadata = json.loads(metadata_bytes.decode("utf-8"))
    return metadata, payload
    raise TimeoutError("timed out waiting for EV-MUX frame")

def send_request(self, channel, method, body):
    seq = self.next_seq
    self.next_seq += 1
    payload = json.dumps(body, separators=(",", ":")).encode("utf-8")
    metadata = {
        "sid": "debug" if channel == "debug.rpc" else "user",
        "seq": seq,
        "channel": channel,
        "type": "req",
        "method": method,
        "encoding": "json",
        "ts_ms": int(time.monotonic() * 1000),
    }
    metadata_bytes = json.dumps(
        metadata, separators=(",", ":")).encode("utf-8")
    crc = zlib.crc32(payload) & 0xFFFFFFFF
    header = "EVMUX/1 h={} p={} c={:08X}\r\n".format(
        len(metadata_bytes), len(payload), crc).encode("ascii")
    self.port.write(SOF + header + metadata_bytes + payload + EOF)
    self.port.flush()
    return seq

def _observe(self, metadata, payload):
    channel = metadata.get("channel")
    if channel in ("repl.stdout", "repl.stderr", "log.idf"):
        self.text_tail.extend(payload)
        del self.text_tail[:-8192]
        text = payload.decode("utf-8", errors="replace")
        sys.stdout.write("[{}] {}".format(channel, text))
        sys.stdout.flush()

def rpc(self, channel, method, body, timeout=10):
    seq = self.send_request(channel, method, body)
    deadline = time.monotonic() + timeout
    while True:
        metadata, payload = self.read_frame(deadline)
        self._observe(metadata, payload)
        if (metadata.get("channel") != channel
            or metadata.get("type") != "rsp"
            or metadata.get("seq") != seq):
            continue
        if metadata.get("encoding") != "json":
            raise RuntimeError("unexpected binary RPC response")
        result = json.loads(payload.decode("utf-8"))
        if not result.get("ok"):
            raise RuntimeError("{} failed: {}".format(method, result))
        return result

```

(continues on next page)

(continued from previous page)

```

def wait_for_text(self, marker, timeout=15):
    deadline = time.monotonic() + timeout
    while marker not in self.text_tail:
        metadata, payload = self.read_frame(deadline)
        self._observe(metadata, payload)
    return bytes(self.text_tail)

def upload_script(mux, path, source):
    content = source.encode("utf-8")
    for offset in range(0, len(content), WRITE_CHUNK):
        chunk = content[offset:offset + WRITE_CHUNK]
        result = mux.rpc("user.rpc", "script.write", {
            "path": path,
            "mode": "overwrite",
            "encoding": "utf-8",
            "contentBase64": base64.b64encode(chunk).decode("ascii"),
            "offset": offset,
            "totalBytes": len(content),
        })
        expected_complete = offset + len(chunk) == len(content)
        if result.get("complete") != expected_complete:
            raise RuntimeError("unexpected script.write completion state")

def main():
    parser = argparse.ArgumentParser(
        description="ESP-VISION EV-MUX USB/USJ closed-loop smoke test")
    parser.add_argument("port", help="USB-OTG CDC or USB-Serial-JTAG port")
    args = parser.parse_args()

    with serial.Serial(
        args.port, baudrate=115200, timeout=0.2, write_timeout=5) as port:
        try:
            port.dtr = True
        except (OSError, serial.SerialException):
            pass
        port.reset_input_buffer()
        time.sleep(0.1)
        mux = EvMux(port)

        hello = mux.rpc("user.rpc", "hello", {})
        if hello.get("firmware", {}).get("id") != "esp-vision":
            raise RuntimeError("connected device is not ESP-VISION")

        capabilities = mux.rpc("user.rpc", "capabilities", {})
        version = capabilities.get("protocol", {}).get("evMuxVersion")
        if version != 3:
            raise RuntimeError("unsupported EV-MUX version: {}".format(version))
        required = {"user.rpc", "debug.rpc", "repl.stdout"}
        if not required.issubset(set(capabilities.get("channels", []))):
            raise RuntimeError("required EV-MUX channels are unavailable")

        state = mux.rpc("debug.rpc", "transport.state", {})
        print("routes: user={user[route]} debug={debug[route]}".format(**state))

        mux.rpc("user.rpc", "device.control", {"action": "stop"})
        mux.wait_for_text(b">>>", timeout=10)

        path = "/evmux_smoke.py"
        upload_script(mux, path, DEVICE_SCRIPT)

```

(continues on next page)

(continued from previous page)

```
    mux.rpc("user.rpc", "script.run", {"path": path})
    mux.wait_for_text(b"EVTEST:OK", timeout=20)
    print("\nPASS: EV-MUX upload, execution, camera capture, and stdout_
↪verified")

if __name__ == "__main__":
    main()
```

This test deliberately treats a timeout, malformed frame, rejected route, negative RPC response, missing camera marker, or unsupported protocol version as a failure. For product integration, retain the same framing and correlation rules, keep reading unrelated events while awaiting an RPC sequence, handle `VM_BUSY` and `VM_TIMEOUT` explicitly, and never report success until the expected device-side output or result has been observed.

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`).

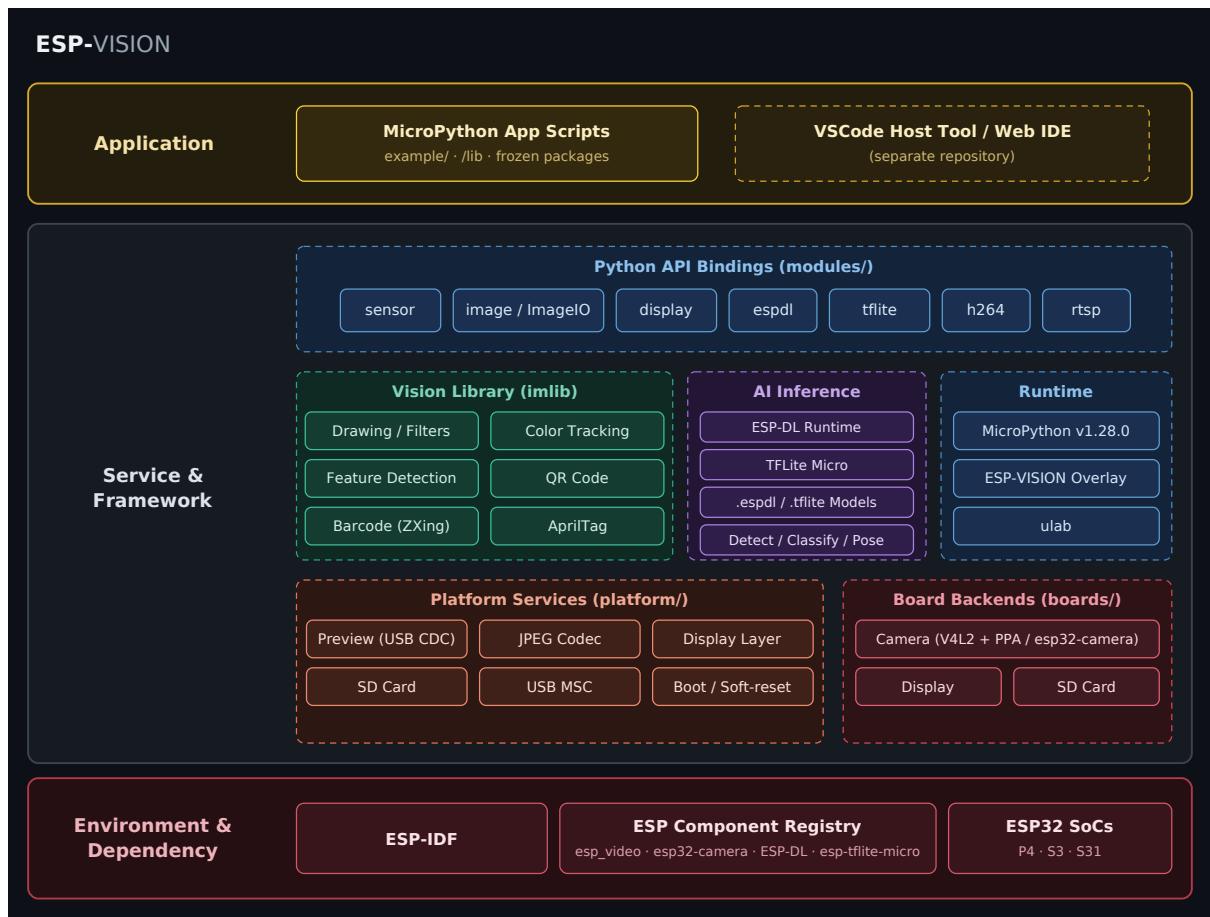
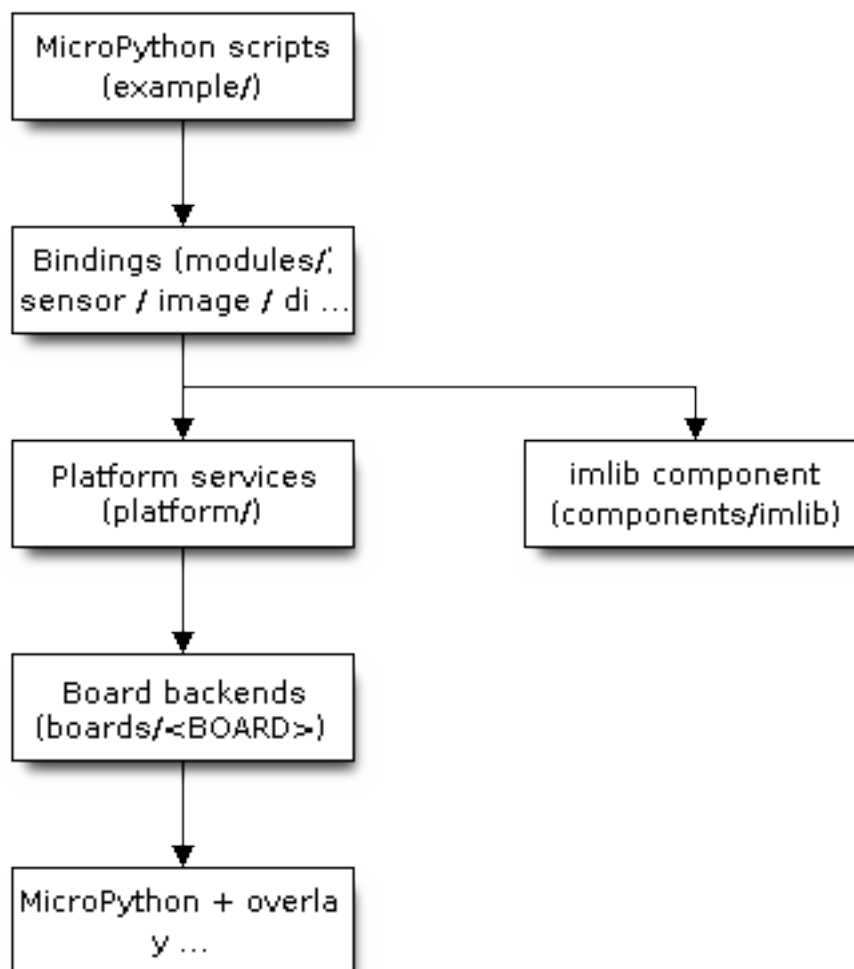


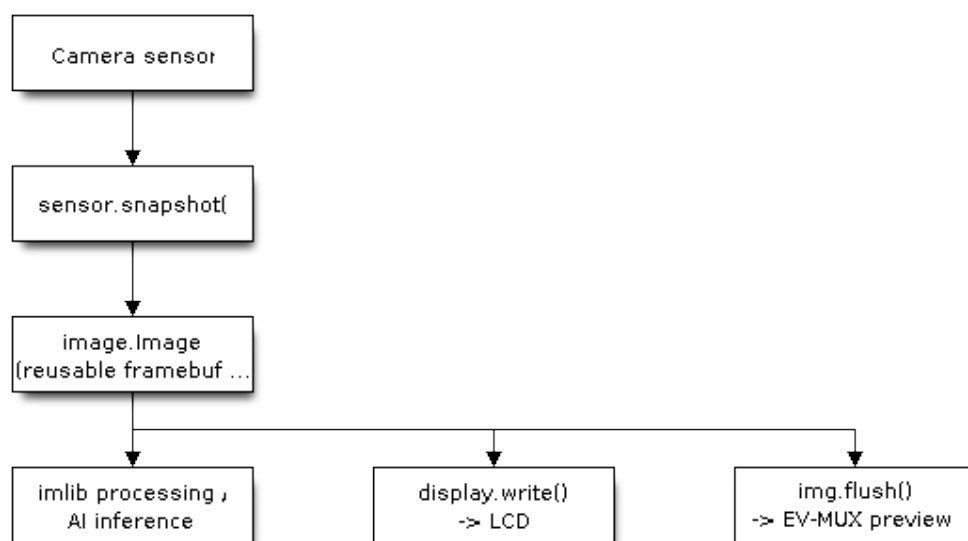
Fig. 1: ESP-VISION layered architecture overview

7.1 Layered Overview



- **Bindings** (modules/): the USER_C_MODULES layer. The main modules image, sensor, display, espdl, and tflite 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. ev_channel.c / ev_mux.c / ev_control_transport.c/ev_stdio.c (EV-MUX / EV-ATP transport), preview.c (EV-MUX JPEG preview), display.c (generic display layer), sdcard.c (mount at /sdcard), usb_msc.c (exposes the ffat partition over TinyUSB MSC), usb_auto_download.c (S3 OTG CDC reset/download protocol), 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`, ESP-DL inference, or TFLite Micro inference on the image, and send it to the LCD with `display.write()` or to the host preview with `img.flush()`.

7.3 Low-Level Transport

ESP-VISION host transport has three layers: physical sinks, routed streams, and EV-MUX logical channels.

7.3.1 Physical Sinks

`platform/ev_channel.c` defines the physical write targets:

sink	Meaning
<code>usj</code>	USB-Serial-JTAG. Available on boards that expose this peripheral; it is the preferred fallback when no host holds USB-OTG CDC open.
<code>cdc</code>	USB-OTG CDC. When available, opening the port (DTR) makes it the automatic route for every stream. It also provides MSC file preview.
<code>uart</code>	Wired auxiliary sink for runtime redirection (<code>route.bind / route.auto</code>); never selected automatically. A stream bound to UART accepts its framed ingress there as well.
<code>console</code>	MicroPython stdout fallback.
<code>null</code>	Drops output; <code>none</code> is an alias.

`cdc` reports two states: `present` means the USB-OTG interface is enumerated; `ready` means a host holds the port open (DTR asserted). `ready` alone decides routing —there is no activation RPC, lease, or heartbeat anywhere in the design.

7.3.2 Routed Streams

`ev_stream_t` is the physical routing abstraction. There are fewer streams than EV-MUX channels. A route applies to a complete stream, not to one logical channel: once a stream is bound to a sink, every channel assigned to that stream uses that sink. Device-to-host frames are written to it, and host-to-device frames for those channels are accepted only from it.

One rule decides the automatic route of every stream: **the active sink is ``cdc`` while a host holds the USB-OTG port open (DTR), otherwise it is the board's preferred fallback (USJ when available, then the console fallback).** A dedicated transport task watches the DTR edge and applies it to both streams; `route.changed` (reason `cdc_connected` / `cdc_disconnected`) is emitted per stream on every transition. Host disconnect or crash drops DTR, so the fallback is automatic —no keepalive is required.

stream	Carries	Routing policy
EV_STREAM_USER	Normal work plane: <code>user.rpc</code> , <code>repl.stdin</code> , <code>repl.signal</code> , <code>repl.stdout</code> , <code>repl.stderr</code> , and <code>preview.frame</code>	Follows the active sink (DTR rule above).
EV_STREAM_DEBUG	Debug and system traffic: <code>debug.rpc</code> / <code>log.idf</code>	Follows the active sink, so the connected USB port carries every logic channel.

`route.bind` pins one stream to a fixed sink (runtime redirection, e.g. `debug` to `uart` for a wired log tap); a manually bound stream leaves the DTR rule until `route.auto` restores it.

7.3.3 EV-MUX Channels

EV-MUX channels describe end-to-end protocol semantics; they are not physical ports. Each channel is assigned to exactly one stream, and a host must not select a sink independently for an individual channel. `direction` describes frame direction. Device-to-host frames use the current sink of their stream; host-to-device frames are authorized against the current sink of their stream.

channel	direction	assigned stream	payload type	Purpose
user.rpc	bidirectional	user (rsp / event)	JSON	Normal host work control such as hello, capabilities, script.write, script.run, and device.control.
debug.rpc	bidirectional	debug (rsp / event)	JSON; debug.capture_frame answers with a binary JPEG payload inside the response frame itself	System/debug EV-ATP requests, responses, events, and errors. req is Host -> Device; rsp / event are Device -> Host.
repl.stdin	Host -> Device	user	text	Host input to the REPL; not an output stream.
repl.signal	Host -> Device	user	small binary/text	Host signals such as Ctrl-C.
repl.stdout	Device -> Host	user	text	Python print(), REPL prompts, and C stdout.
repl.stderr	Device -> Host	user	text	Python exceptions and C stderr.
log.idf	Device -> Host	debug	text	ESP-IDF ESP_LOGx output.
preview.frame	Device -> Host	user	binary JPEG	Preview frames produced continuously by img.flush(); no subscription is needed and no RPC is involved.

7.3.4 EV-MUX Frame Format

All EV-MUX frames are length-prefixed, and payloads may be binary:

```
\x1eEVMUX/1 h=<metadata_len> p=<payload_len> c=<crc32>\r\n
<metadata JSON bytes>
<payload bytes>
\x1f
```

h and p are authoritative lengths; 0x1f is only an EOF guard and resynchronization aid. Firmware-generated frames include a payload CRC32. Host-to-device command frames may use c=00000000; the firmware receiver treats zero as “skip CRC validation”.

7.3.5 EV-ATP Control

EV-ATP RPC is split by semantics:

- user.rpc carries normal IDE operation: hello, capabilities, script.write, script.run, and device.control.
- debug.rpc carries system/debug commands: transport.state, route.get, route.bind, route.auto, debug.info, and debug.capture_frame. Responses return to the sink the request arrived on. debug.capture_frame returns its image as the binary payload of the debug.rpc response frame (contentType=image/jpeg plus width / height in metadata); there is no separate data channel.

route.changed is a routing event on user.rpc / user; it is not a debug.rpc event. It reports a stream-level sink change (reason cdc_connected / cdc_disconnected), so the host must update the connection state of every channel assigned to that stream.

Routing itself needs no RPC: opening the USB-OTG port makes it the active sink, and closing it selects the board's fallback. There is intentionally no activation, lease, or heartbeat method —the wire state (DTR) is the whole routing protocol.

7.3.6 Host Integration Contract

The host uses exactly one physical USB connection at a time. On dual-USB boards, USB-OTG and USJ are equivalent choices and either one carries every channel. The connection sequence is:

1. Open the chosen port. EV-MUX is enabled at boot, so no REPL access is required: the device emits a bootstrap `hello` event when a USB link appears (and again after every soft reset).
2. Send a framed `hello` request (or consume the boot event), then request capabilities. Verify that `firmware.id` is `esp-vision`, check `evMuxVersion`, and negotiate the advertised features/channels.
3. Use the device: REPL channels, `script.write / script.run`, `preview`, `debug.rpc`, and `log.idf` are all live on the opened port. No activation step and no heartbeat exist; the port is the route.
4. On disconnect just close the port. If USB-OTG closes (or the cable drops), all streams select the board's fallback automatically and a `route.changed` event reports it.

`sensor.ev mux()` remains as a debug toggle (for example to recover a plain text REPL with `sensor.ev mux(False)`); boards can opt out of boot-time enablement by defining `ESP_VISION_EV_MUX_DEFAULT_ENABLED` to 0 in `boardconfig.h`.

The `hello`, `capabilities`, and `debug.info` device payloads all contain the same `firmware` object. Its stable `id` is `esp-vision`. Its version is the `ESP-IDF PROJECT_VER` derived from Git: an official release build reports its `YYYY.MM.DD` tag, while an untagged development build includes the `-N-g<commit>` suffix (and may include `-dirty`). Hosts may display or compare this build identity for update UX, but protocol compatibility must be decided from `evMuxVersion` and advertised capabilities rather than the release string.

The host demultiplexer must inspect `metadata.channel` on every frame and dispatch by channel. It must not infer frame type from the port or a preceding frame. A host should maintain a static `channel -> stream` table and a dynamic `stream -> sink` table; route changes update only the latter.

7.3.7 Receive-Side Demultiplexing

The firmware receiver keeps state only for byte-stream frame assembly. Once a complete EV-MUX frame arrives, its assigned stream is first validated against the actual ingress sink, then it is dispatched immediately by `metadata.channel / type / method`:

- `repl.stdin` is queued into the REPL input ring buffer.
- `repl.signal` schedules signals such as keyboard interrupt.
- `user.rpc / debug.rpc` first select the RPC domain, then execute EV-ATP control logic by `method`.

Ingress authorization is a single whitelist: the discovery methods `hello / capabilities` are accepted on either USB sink; every other channel is accepted only from the current route of its assigned stream. This also permits framed control on UART after an explicit `route.bind`. Business logic must not infer frame type from a “current mode”.

7.3.8 Transport Execution Model

EV-MUX is enabled at boot by default (`ESP_VISION_EV_MUX_DEFAULT_ENABLED`), so the control plane is always discoverable without REPL access; after every soft reset the mux state and routes return to the same deterministic defaults.

Frame reception does not depend on the REPL. A dedicated transport task (`ev_transport`, see `platform/ev_control_transport.c`) owns the receive path: it pumps the existing TinyUSB CDC device stack, drains the USJ/CDC/UART ingress ring buffers into the per-ingress frame parsers, dispatches complete frames, and applies the DTR routing rule (active sink switch and `route.changed` emission). `mp_hal_stdin_rx_chr()` no longer parses frames; when EV-MUX is enabled it only consumes framed `repl.stdin` bytes.

Dispatch is split by execution context:

- Transport-task RPCs (answered immediately, even while user code runs): `hello`, `capabilities`, `transport.state`, `route.*`, and `device.control`. `repl.signal` schedules the keyboard interrupt from the transport task, so host Ctrl-C reaches a running script.
- VM-task RPCs (queued to the interpreter and executed from `mp_hal_stdin_rx_chr()` context): `script.write`, `script.run`, `debug.info`, and `debug.capture_frame` —anything touching MicroPython objects, the VFS, or the camera. A full queue is answered with a `VM_BUSY` error.
- `repl.stdin` payloads are queued into the framed REPL input ring and consumed by the REPL loop.

While EV-MUX is enabled, its low-level USJ/CDC/UART receive paths parse frames only: they do not interpret `0x03` as Ctrl-C (interrupt semantics belong to the framed `repl.signal` channel), and unframed bytes are discarded by the frame parser's SOF resynchronization. Raw bytes emitted by the IDF system console before protocol initialization or on fatal paths are the exception described below.

7.3.9 IDF Console and Interface Selection

The ESP-IDF system console and ESP-VISION interface selection are configured independently. Board `sdkconfig` files do not override `CONFIG_ESP_CONSOLE_*` or `CONFIG_USJ_ENABLE_USB_SERIAL_JTAG`; console routing, bootloader output, and system debugging capabilities use the defaults for the current ESP-IDF target and version. The default secondary console on some targets may use USB-Serial-JTAG. That is IDF system-console behavior, not the switch that enables the ESP-VISION USJ interface.

The MicroPython USJ interface is controlled only by `MICROPY_HW_ESP_USB_SERIAL_JTAG` in each board's `mpconfigboard.h`: P4 boards explicitly set it to 1, while S3/S31 boards set it to 0. When enabled, `usb_serial_jtag_init` follows the ESP-IDF USJ driver initialization sequence to enable the USJ bus clock and select/enable the internal PHY explicitly, so it does not depend on a console Kconfig option incidentally keeping the peripheral active. When disabled, the ESP-VISION USJ path is neither compiled nor initialized, although IDF may still use the peripheral according to its default console configuration.

ESP32-S3 boards separately enable `MICROPY_HW_USB_CDC_DTR_RTS_BOOTLOADER` in `mpconfigboard.h` and assign the OTG CDC device PID `0x1001`. The existing MicroPython/EV-MUX TinyUSB CDC stack starts immediately after VM initialization and before camera initialization, filesystem recovery, and `boot.py`; the EV-MUX transport task services that same stack independently while user code runs and always delays by at least one FreeRTOS tick. No second USB driver or descriptor owner is installed, CDC/MSC selection remains in `mpconfigboard.h`, and the IDF system console remains unchanged. PID `0x1001` makes `esptool` select its USB-Serial/JTAG reset sequence; firmware implements the matching DTR/RTS state table in software, although the OTG device does not expose a JTAG interface. A DTR/RTS bootloader request is recorded by the TinyUSB callback and applied by a deferred timer task, outside the control-request callback. On S3, that task independently follows MicroPython's ROM contract (`usb_usj_mode`, `usb_dc_prepare_persist`, and `USBDC_BOOT_DFU`), then the shutdown handler sets `RTC_CNTL_FORCE_DOWNLOAD_BOOT` and calls `esp_restart`. It does not wait for a host bus reset or tear down the TinyUSB controller from the callback. The host observes a normal re-enumeration and must reopen the ROM CDC node; on Linux, `esptool 5.3` can report EIO from its stale first handle even though the ROM loader is active, and the operation can be retried on the new node with `--before no-reset`. Other boards leave this feature disabled by default.

During normal runtime, `ev_stdio_init0` uses `esp_log_set_vprintf` and `newlib stdout/stderr` hooks to wrap output in EV-MUX frames. Boot output sent before the hooks are installed, and fatal-path output that bypasses them, may still appear as raw bytes on UART or USJ according to the system-console defaults. Those bytes are outside the active EV-MUX contract; the host discards them and resynchronizes at a later SOF.

When the ROM loader was entered manually or remains there after a flash, use `esptool --after watchdog-reset` for the final reset: USB-Serial/JTAG's default core reset does not re-sample GPIO0, while the watchdog path performs a full reset back to SPI boot.

7.3.10 Transport Reliability and Observability

- Each frame segment uses a write-all loop with progress detection and a 1500 ms total budget (`platform/ev_mux.c`). Low-level sinks return their actual byte count; failure to make further progress or a timeout

aborts the send, increments `txTimeout`, and marks the sink congested. Bytes already written before a failure cannot be rolled back, so the host parser must discard truncated data and resynchronize at a later valid SOF; the guarantee is bounded, observable failure, not an atomic partial write.

- `preview.frame` is the only class that may be dropped proactively. During the 500 ms cooldown after sink congestion, subsequent previews are dropped whole before encoding or writing and increment `txDrop`; the first frame after cooldown probes for recovery. The USJ route is also capped at one frame per 100 ms. RPC and REPL are never pre-dropped for congestion, although a real sink write failure can still make the current send return an error.
- The USJ, CDC, and UART ingress paths each have a 2048-byte physical input ring. These rings are staging buffers drained continuously by the parser, not a requirement that a protocol frame fit in the ring. The pure-C `debug.info scope transport.stats` remains available while the VM is busy and reports RX byte counts, ring-full events, complete/malformed/rejected frames, and the `replRx`, `vmBusy`, `txTimeout`, and `txDrop` counters. `replRx` is a separate 512-byte REPL queue; overflow is currently counted but does not emit a separate asynchronous host event.

7.4 Source Tree

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

7.5 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.

7.6 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.

7.7 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, TFLite Micro 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 / tflite / h264 / rtsp
|
+-- Algorithms and middleware
|   +-- OpenMV imlib subset
|   +-- ulab / ZXing-C++ / ESP-DL / TFLite Micro
```

(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	<code>lib/micropython</code>	Python runtime and ESP32 port baseline	MIT
micropython-ulab 6.12	<code>lib/ulab</code>	Array and numerical computing	MIT
OpenMV v5.0.0 <code>imlib</code> subset	<code>components/imlib</code>	Image processing and drawing algorithms	MIT, with separately licensed files
OpenMV Python binding code	<code>modules/py_image*</code> , <code>modules/py_helper*</code>	OpenMV-style image API bindings	MIT
OpenMV AprilTag implementation	<code>components/imlib/upstream/apriltag.c</code>	AprilTag and rectangle detection	BSD-2-Clause
ZXing-C++ v3.0.2	<code>lib/zxing-cpp</code>	1D barcode reader backend	Apache-2.0
ESP-DL	Component Registry	Model inference runtime	MIT
TensorFlow Lite Micro / <code>esp-tflite-micro</code>	Component Registry	TensorFlow Lite Micro inference runtime	Apache-2.0
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

d

`display`, [59](#)

e

`espdl`, [62](#)

h

`h264`, [69](#)

i

`image`, [42](#)

`imageio`, [67](#)

r

`rtsp`, [71](#)

s

`sensor`, [39](#)

t

`tflite`, [65](#)

Index

A

`a_bins()` (*image.histogram method*), 50
`a_lq()` (*image.statistics method*), 52
`a_max()` (*image.statistics method*), 52
`a_mean()` (*image.statistics method*), 52
`a_median()` (*image.statistics method*), 52
`a_min()` (*image.statistics method*), 52
`a_mode()` (*image.statistics method*), 52
`a_stdev()` (*image.statistics method*), 52
`a_uq()` (*image.statistics method*), 52
`a_value()` (*image.percentile method*), 51
`a_value()` (*image.threshold method*), 51
`APPLY_COLOR_PALETTE_FIRST` (in module *image*), 43
`apriltag` (class in *image*), 50
`AREA` (in module *image*), 43
`area()` (*image.blob method*), 48

B

`b_bins()` (*image.histogram method*), 50
`b_lq()` (*image.statistics method*), 52
`b_max()` (*image.statistics method*), 52
`b_mean()` (*image.statistics method*), 52
`b_median()` (*image.statistics method*), 52
`b_min()` (*image.statistics method*), 52
`b_mode()` (*image.statistics method*), 52
`b_stdev()` (*image.statistics method*), 52
`b_uq()` (*image.statistics method*), 52
`b_value()` (*image.percentile method*), 51
`b_value()` (*image.threshold method*), 51
`backlight()` (*display.ESP32Display method*), 59
`barcode` (class in *image*), 50
`BAYER` (in module *image*), 43
`BICUBIC` (in module *image*), 43
`bilateral()` (*image.Image method*), 56
`BILINEAR` (in module *image*), 43
`BINARY` (in module *image*), 43
`binary()` (*image.Image method*), 55
`binary_to_grayscale()` (in module *image*), 45
`binary_to_lab()` (in module *image*), 45
`binary_to_rgb()` (in module *image*), 45
`binary_to_yuv()` (in module *image*), 45
`bins()` (*image.histogram method*), 50
`BLACK_BACKGROUND` (in module *image*), 44
`blob` (class in *image*), 47
`blur()` (*image.Image method*), 56
`buffer_size()` (*imageio.ImageIO method*), 68

`bytearray()` (*image.Image method*), 53

C

`CENTER` (in module *image*), 43
`circle` (class in *image*), 47
`circle()` (*image.circle method*), 47
`classify()` (*espdl.ImageNetCls method*), 64
`clear()` (*display.ESP32Display method*), 59
`clear()` (*image.Image method*), 54
`close()` (*h264.H264Encoder method*), 70
`close()` (*image.Image method*), 55
`close()` (*imageio.ImageIO method*), 68
`CODABAR` (in module *image*), 45
`code()` (*image.blob method*), 48
`CODE128` (in module *image*), 45
`CODE39` (in module *image*), 45
`CODE93` (in module *image*), 45
`compactness()` (*image.blob method*), 48
`compress()` (*image.Image method*), 53
`convexity()` (*image.blob method*), 48
`copy()` (*image.Image method*), 53
`CORNER_AGAST` (in module *image*), 44
`CORNER_FAST` (in module *image*), 44
`corners()` (*image.barcode method*), 50
`corners()` (*image.blob method*), 47
`corners()` (*image.qrcode method*), 49
`corners()` (*image.rect method*), 47
`count()` (*image.blob method*), 48
`count()` (*imageio.ImageIO method*), 67
`crop()` (*image.Image method*), 53
`cx()` (*image.blob method*), 48
`cxr()` (*image.blob method*), 48
`cy()` (*image.blob method*), 48
`cyf()` (*image.blob method*), 48

D

`data_type()` (*image.qrcode method*), 49
`DATABAR` (in module *image*), 45
`DATABAR_EXP` (in module *image*), 45
`deinit()` (*display.ESP32Display method*), 59
`deinit()` (*espdl.ESPDet method*), 63
`deinit()` (*espdl.ImageNetCls method*), 64
`deinit()` (*espdl.Model method*), 62
`deinit()` (*espdl.PedestrianDetect method*), 63
`deinit()` (*espdl.YOLO11 method*), 63
`deinit()` (*espdl.YOLO11nPose method*), 64
`deinit()` (*tflite.Model method*), 66

[density\(\)](#) (*image.blob method*), 48
[detect\(\)](#) (*espd.ESPDet method*), 63
[detect\(\)](#) (*espd.PedestrianDetect method*), 63
[detect\(\)](#) (*espd.YOLO11 method*), 63
[detect\(\)](#) (*espd.YOLO11nPose method*), 64
[difference\(\)](#) (*image.Image method*), 55
[dilate\(\)](#) (*image.Image method*), 55
[display](#)
 module, 59
[draw_arrow\(\)](#) (*image.Image method*), 54
[draw_circle\(\)](#) (*image.Image method*), 54
[draw_cross\(\)](#) (*image.Image method*), 54
[draw_edges\(\)](#) (*image.Image method*), 54
[draw_ellipse\(\)](#) (*image.Image method*), 54
[draw_image\(\)](#) (*image.Image method*), 54
[draw_keypoints\(\)](#) (*image.Image method*), 54
[draw_line\(\)](#) (*image.Image method*), 54
[draw_rectangle\(\)](#) (*image.Image method*), 54
[draw_string\(\)](#) (*image.Image method*), 54

E

[EAN13](#) (*in module image*), 45
[EAN2](#) (*in module image*), 44
[EAN5](#) (*in module image*), 44
[EAN8](#) (*in module image*), 44
[ecc_level\(\)](#) (*image.qrcode method*), 49
[eci\(\)](#) (*image.qrcode method*), 49
[EDGE_CANNY](#) (*in module image*), 44
[EDGE_SIMPLE](#) (*in module image*), 44
[elongation\(\)](#) (*image.blob method*), 48
[enclosed_ellipse\(\)](#) (*image.blob method*), 49
[enclosing_circle\(\)](#) (*image.blob method*), 49
[encode\(\)](#) (*h264.H264Encoder method*), 70
[erode\(\)](#) (*image.Image method*), 55
[ESP32Display](#) (*class in display*), 59
[ESPDet](#) (*class in espd*), 62
[espd](#)
 module, 62
[extent\(\)](#) (*image.blob method*), 48
[EXTRACT_RGB_CHANNEL_FIRST](#) (*in module image*), 43

F

[FILE_STREAM](#) (*in module imageio*), 67
[find_apriltags\(\)](#) (*image.Image method*), 58
[find_barcodes\(\)](#) (*image.Image method*), 58
[find_blobs\(\)](#) (*image.Image method*), 57
[find_circles\(\)](#) (*image.Image method*), 57
[find_line_segments\(\)](#) (*image.Image method*), 57
[find_lines\(\)](#) (*image.Image method*), 57
[find_qrcodes\(\)](#) (*image.Image method*), 58
[find_rects\(\)](#) (*image.Image method*), 57
[flush\(\)](#) (*image.Image method*), 53
[format\(\)](#) (*image.Image method*), 53

G

[gaussian\(\)](#) (*image.Image method*), 56

[gaussian_blur\(\)](#) (*image.Image method*), 56
[get_framesize\(\)](#) (*in module sensor*), 40
[get_hist\(\)](#) (*image.Image method*), 56
[get_histogram\(\)](#) (*image.Image method*), 56
[get_hmirror\(\)](#) (*in module sensor*), 40
[get_id\(\)](#) (*in module sensor*), 40
[get_percentile\(\)](#) (*image.histogram method*), 50
[get_pixel\(\)](#) (*image.Image method*), 53
[get_pixformat\(\)](#) (*in module sensor*), 40
[get_regression\(\)](#) (*image.Image method*), 57
[get_statistics\(\)](#) (*image.histogram method*), 51
[get_statistics\(\)](#) (*image.Image method*), 57
[get_stats\(\)](#) (*image.histogram method*), 51
[get_stats\(\)](#) (*image.Image method*), 57
[get_threshold\(\)](#) (*image.histogram method*), 50
[get_vflip\(\)](#) (*in module sensor*), 40
[GRAYSCALE](#) (*in module image*), 43
[GRAYSCALE](#) (*in module sensor*), 39
[grayscale_to_binary\(\)](#) (*in module image*), 45
[grayscale_to_lab\(\)](#) (*in module image*), 45
[grayscale_to_rgb\(\)](#) (*in module image*), 45
[grayscale_to_yuv\(\)](#) (*in module image*), 45

H

[h\(\)](#) (*image.barcode method*), 50
[h\(\)](#) (*image.blob method*), 48
[h\(\)](#) (*image.qrcode method*), 49
[h\(\)](#) (*image.rect method*), 47
[h264](#)
 module, 69
[H264Encoder](#) (*class in h264*), 69
[height\(\)](#) (*display.ESP32Display method*), 59
[height\(\)](#) (*image.Image method*), 52
[height\(\)](#) (*in module sensor*), 40
[histeq\(\)](#) (*image.Image method*), 55
[histogram](#) (*class in image*), 50
[histogram\(\)](#) (*image.Image method*), 56
[HMIRROR](#) (*in module image*), 43

I

[I25](#) (*in module image*), 45
[image](#)
 module, 42
[Image](#) (*class in image*), 52
[imageio](#)
 module, 67
[ImageIO](#) (*class in imageio*), 67
[ImageNetCls](#) (*class in espd*), 64
[inputs\(\)](#) (*espd.Model method*), 62
[invert\(\)](#) (*image.Image method*), 55
[is_alphanumeric\(\)](#) (*image.qrcode method*), 49
[is_binary\(\)](#) (*image.qrcode method*), 49
[is_closed\(\)](#) (*imageio.ImageIO method*), 67
[is_kanji\(\)](#) (*image.qrcode method*), 50
[is_numeric\(\)](#) (*image.qrcode method*), 49
[ISBN10](#) (*in module image*), 44
[ISBN13](#) (*in module image*), 45

J

JPEG (in module image), 43
JPEG_SUBSAMPLING_420 (in module image), 44
JPEG_SUBSAMPLING_422 (in module image), 44
JPEG_SUBSAMPLING_444 (in module image), 44
JPEG_SUBSAMPLING_AUTO (in module image), 44

K

keyframe() (h264.H264Encoder method), 70

L

l_bins() (image.histogram method), 50
l_lq() (image.statistics method), 52
l_max() (image.statistics method), 52
l_mean() (image.statistics method), 51
l_median() (image.statistics method), 52
l_min() (image.statistics method), 52
l_mode() (image.statistics method), 52
l_stdev() (image.statistics method), 52
l_uq() (image.statistics method), 52
l_value() (image.percentile method), 51
l_value() (image.threshold method), 51
lab_to_binary() (in module image), 46
lab_to_grayscale() (in module image), 46
lab_to_rgb() (in module image), 46
lab_to_yuv() (in module image), 46
laplacian() (image.Image method), 56
length() (image.line method), 46
line (class in image), 46
line() (image.line method), 46
load_model() (in module espdl), 62
lq() (image.statistics method), 51

M

magnitude() (image.circle method), 47
magnitude() (image.line method), 46
magnitude() (image.rect method), 47
major_axis_line() (image.blob method), 49
mask() (image.qrcode method), 49
max() (image.statistics method), 51
mean() (image.Image method), 55
mean() (image.statistics method), 51
median() (image.Image method), 55
median() (image.statistics method), 51
MEMORY_STREAM (in module imageio), 67
midpoint() (image.Image method), 56
min() (image.statistics method), 51
min_corners() (image.blob method), 47
minor_axis_line() (image.blob method), 49
mode() (image.Image method), 56
mode() (image.statistics method), 51
Model (class in espdl), 62
Model (class in tfliite), 66
module
 display, 59
 espdl, 62
 h264, 69

image, 42
imageio, 67
rtsp, 71
sensor, 39
tflite, 65

morph() (image.Image method), 56

O

offset() (imageio.ImageIO method), 67
open() (image.Image method), 55
outputs() (espdl.Model method), 62

P

PALETTE_DEPTH (in module image), 43
PALETTE_EVT_DARK (in module image), 43
PALETTE_EVT_LIGHT (in module image), 43
PALETTE_IRONBOW (in module image), 43
PALETTE_RAINBOW (in module image), 43
payload() (image.barcode method), 50
payload() (image.qrcode method), 49
PDF417 (in module image), 45
PedestrianDetect (class in espdl), 63
percentile (class in image), 51
perimeter() (image.blob method), 48
pixels() (image.blob method), 48
PNG (in module image), 43
predict() (espdl.Model method), 62
predict() (tflite.Model method), 66

Q

QQVGA (in module sensor), 39
qrcode (class in image), 49
quality() (image.barcode method), 50
QVGA (in module sensor), 39

R

r() (image.circle method), 47
read() (imageio.ImageIO method), 68
rect (class in image), 47
rect() (image.barcode method), 50
rect() (image.blob method), 47
rect() (image.qrcode method), 49
rect() (image.rect method), 47
reset() (in module sensor), 40
RGB565 (in module image), 43
RGB565 (in module sensor), 39
rgb_to_binary() (in module image), 45
rgb_to_grayscale() (in module image), 45
rgb_to_lab() (in module image), 46
rgb_to_yuv() (in module image), 46
rho() (image.line method), 46
ROTATE_180 (in module image), 44
ROTATE_270 (in module image), 44
ROTATE_90 (in module image), 44
rotation() (image.barcode method), 50
rotation() (image.blob method), 48
rotation_deg() (image.blob method), 48

`rotation_rad()` (*image.blob method*), 48
`roundness()` (*image.blob method*), 48
`rtsp`
 module, 71
`RTSPServer` (class in *rtsp*), 71

S

`save()` (*image.Image method*), 53
`scale()` (*image.Image method*), 53
`SCALE_ASPECT_EXPAND` (in module *image*), 44
`SCALE_ASPECT_IGNORE` (in module *image*), 44
`SCALE_ASPECT_KEEP` (in module *image*), 44
`SEARCH_DS` (in module *image*), 44
`SEARCH_EX` (in module *image*), 44
`seek()` (*imageio.ImageIO method*), 68
`send()` (*rtsp.RTSPServer method*), 71
`sensor`
 module, 39
`SensorStatus` (class in *sensor*), 41
`set_framesize()` (in module *sensor*), 40
`set_hmirror()` (in module *sensor*), 40
`set_pixel()` (*image.Image method*), 53
`set_pixformat()` (in module *sensor*), 40
`set_thresholds()` (*espd.ESPDet method*), 63
`set_thresholds()` (*espd.ImageNetCls method*), 64
`set_thresholds()` (*espd.PedestrianDetect method*), 63
`set_thresholds()` (*espd.YOLO11 method*), 64
`set_thresholds()` (*espd.YOLO11nPose method*), 64
`set_vflip()` (in module *sensor*), 40
`shutdown()` (in module *sensor*), 40
`size()` (*image.Image method*), 53
`size()` (*imageio.ImageIO method*), 68
`skip_frames()` (in module *sensor*), 40
`snapshot()` (in module *sensor*), 40
`solidity()` (*image.blob method*), 48
`statistics` (class in *image*), 51
`statistics()` (*image.histogram method*), 51
`statistics()` (*image.Image method*), 57
`status()` (in module *sensor*), 41
`stdev()` (*image.statistics method*), 51
`stop()` (*rtsp.RTSPServer method*), 71
`sync()` (*imageio.ImageIO method*), 68

T

`TAG16H5` (in module *image*), 45
`TAG25H7` (in module *image*), 45
`TAG25H9` (in module *image*), 45
`TAG36H10` (in module *image*), 45
`TAG36H11` (in module *image*), 45
`tflite`
 module, 65
`theta()` (*image.line method*), 46
`threshold` (class in *image*), 51
`to_bitmap()` (*image.Image method*), 53
`to_grayscale()` (*image.Image method*), 53
`to_ironbow()` (*image.Image method*), 54

`to_jpeg()` (*image.Image method*), 54
`to_png()` (*image.Image method*), 54
`to_rainbow()` (*image.Image method*), 54
`to_rgb565()` (*image.Image method*), 54
`TRANSPPOSE` (in module *image*), 43
`type()` (*image.barcode method*), 50
`type()` (*imageio.ImageIO method*), 67

U

`UPCA` (in module *image*), 45
`UPCE` (in module *image*), 44
`uq()` (*image.statistics method*), 51

V

`value()` (*image.percentile method*), 51
`value()` (*image.threshold method*), 51
`version()` (*image.qrcode method*), 49
`version()` (*imageio.ImageIO method*), 68
`VFLIP` (in module *image*), 43
`VGA` (in module *sensor*), 39

W

`w()` (*image.barcode method*), 50
`w()` (*image.blob method*), 47
`w()` (*image.qrcode method*), 49
`w()` (*image.rect method*), 47
`width()` (*display.ESP32Display method*), 59
`width()` (*image.Image method*), 52
`width()` (in module *sensor*), 40
`write()` (*display.ESP32Display method*), 59
`write()` (*imageio.ImageIO method*), 68

X

`x()` (*image.barcode method*), 50
`x()` (*image.blob method*), 47
`x()` (*image.circle method*), 47
`x()` (*image.qrcode method*), 49
`x()` (*image.rect method*), 47
`x1()` (*image.line method*), 46
`x2()` (*image.line method*), 46
`x_hist_bins()` (*image.blob method*), 48

Y

`y()` (*image.barcode method*), 50
`y()` (*image.blob method*), 47
`y()` (*image.circle method*), 47
`y()` (*image.qrcode method*), 49
`y()` (*image.rect method*), 47
`y1()` (*image.line method*), 46
`y2()` (*image.line method*), 46
`y_hist_bins()` (*image.blob method*), 49
`YOLO11` (class in *espd*), 63
`YOLO11nPose` (class in *espd*), 64
`YUV422` (in module *image*), 43
`yuv_to_binary()` (in module *image*), 46
`yuv_to_grayscale()` (in module *image*), 46
`yuv_to_lab()` (in module *image*), 46
`yuv_to_rgb()` (in module *image*), 46