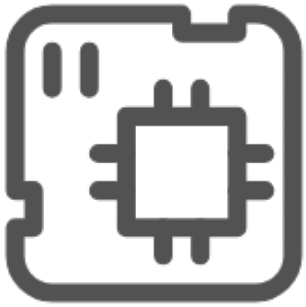
ESP-Brookesia Programming Guide

Release master
Espressif Systems
Apr 13, 2026

Table of contents

Index **217**

ESP-BROOKESIA

		
Getting Started	Utils	HAL
		
Service	Agent	Expression

Overview

ESP-Brookesia is a human-machine interaction development framework for AIoT devices. It streamlines application development and AI capability integration. Built on ESP-IDF and a component-based architecture, it provides full-stack support from hardware abstraction and system services to AI agents, accelerating time-to-market for HMI and AI products.

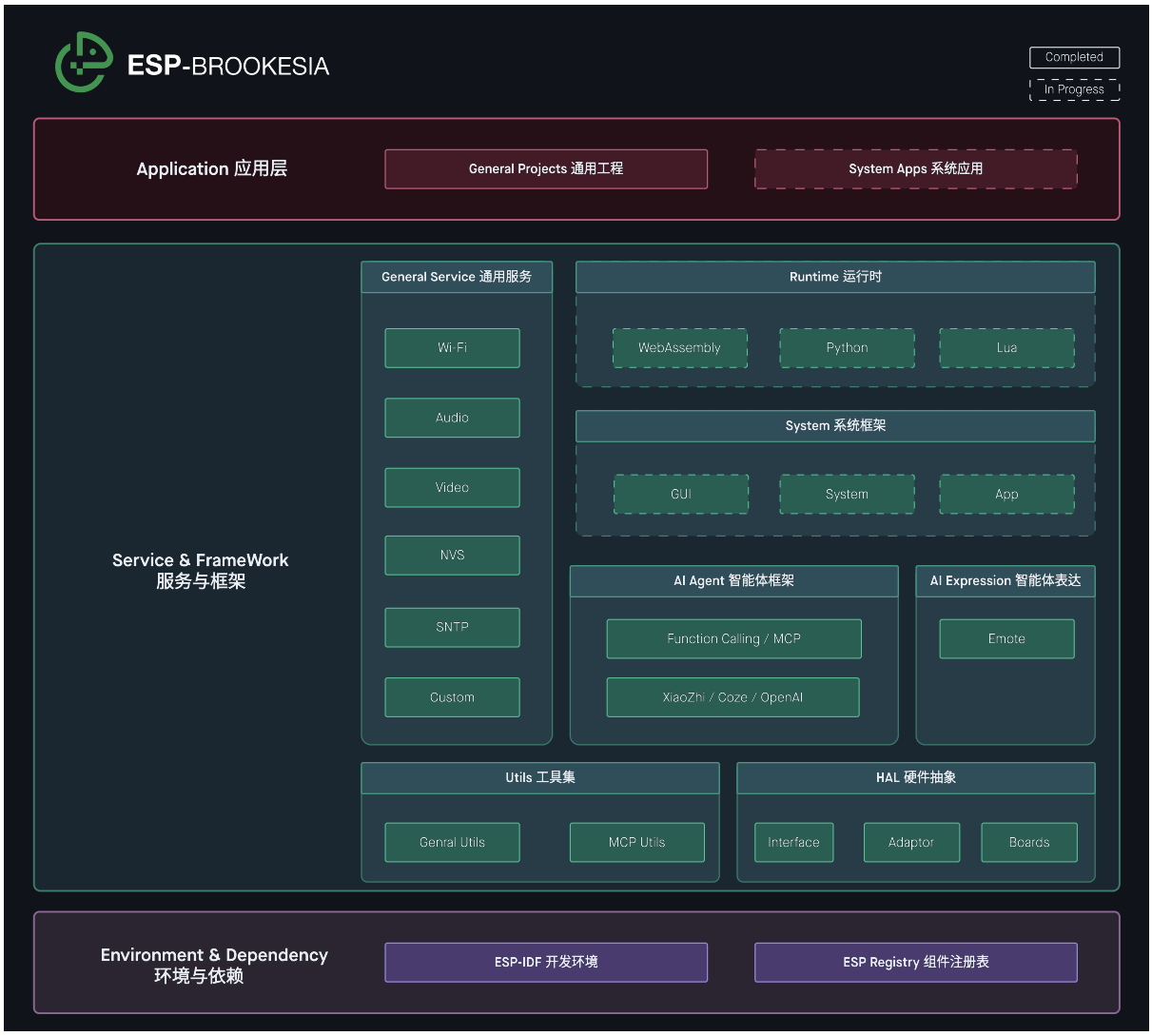
Note: "Brookesia" is a genus of chameleons known for camouflage and adaptation—goals aligned with ESP-Brookesia. The framework aims to offer a flexible, scalable solution that adapts to diverse hardware and application needs, with high adaptability and flexibility like its namesake.

Key features of ESP-Brookesia:

- **Native ESP-IDF integration**: C/C++ development deeply integrated with ESP-IDF and the ESP Registry component catalog, leveraging Espressif ' s open-source ecosystem.
- **Extensible hardware abstraction**: Unified hardware interfaces (audio, display, touch, storage) with board-level adaptation for fast porting.
- **Rich system services**: Wi-Fi, audio/video, using a Manager + Helper architecture for decoupling and extension, providing support for Agent CLI.
- **Multi-LLM backends**: Built-in adapters for OpenAI, Coze, XiaoZhi, and other platforms, with unified agent lifecycle management.
- **MCP protocol support**: Function Calling / MCP exposes device services to large language models for unified LLM–service communication.
- **AI expression**: Emoji sets, animations, and visual feedback for anthropomorphic interaction.

Architecture

ESP-Brookesia uses a layered design with three levels—**environment & dependencies**, **service & framework**, and **application**—as shown below.



Environment & dependencies

The runtime foundation. **ESP-IDF** provides the toolchain, RTOS, and peripheral drivers; **ESP Registry** manages distribution and versioning of framework components and third-party dependencies.

Service & framework

The core layer between the environment and applications. It exposes standardized service interfaces to applications and AI agents, covering utilities, HAL, system services, AI agents, and expression.

- **Utils**: General utilities (logging, checks, state machine, task scheduler, plugins, profilers) and **MCP Utils**, bridging Brookesia services and the MCP engine so registered service functions become standard MCP tools for LLMs.
- **HAL**: **Interface** defines audio, display, touch, status LED, and storage APIs; **Adaptor** provides board-specific implementations and resource mapping; **Boards** provides board-level YAML configuration describing the peripheral topology, pin assignments, and driver parameters.
- **General Service**: Wi-Fi, Audio, Video, NVS, SNTP, and **Custom** extensions. All services use Manager + Helper with local calls and RPC.
- **AI Agent**: Unified agent management with adapters for **Coze**, **OpenAI**, **XiaoZhi**, and **Function Calling / MCP** for bidirectional LLM–service communication.
- **AI Expression**: Visual expression including **Emote** sets and animation control for anthropomorphic UIs.
- **System framework** *(planned)*: GUI, system shell, and app frameworks for phones, speakers, robots, and similar products.
- **Runtime** *(planned)*: WebAssembly, Python, and Lua for dynamic loading and execution.

Application layer

Products and projects built on the layers above:

- **General Projects**: Product-oriented templates integrating framework components for direct product development.
- **System Apps** *(planned)*: Product-oriented system apps such as Settings, AI assistant, and app store, optional and integrable as needed.

Chapter 1

Getting Started

This guide explains how to obtain and use ESP-Brookesia components and how to build and run example projects.

1.1 ESP-Brookesia versioning

From **v0.7**, ESP-Brookesia is componentized. Obtain components via the component registry as follows:

1. Components evolve independently but share the same **major.minor** version and depend on the same ESP-IDF release.
2. The **release** branch maintains historical major versions; **master** integrates new features.

Version support:

Table 1: ESP-Brookesia version support

ESP-Brookesia	ESP-IDF	Main changes	Status
master (v0.7)	>= v5.5, < 6.0	Component manager support	Active development
release/v0.6	>= v5.3, <= 5.5	Preview system framework; ESP-VoCat firmware project	End of maintenance

1.2 Development environment

ESP-IDF is Espressif ' s framework for ESP series chips:

- Libraries and headers provide core building blocks for ESP SoC software.
- Tools for build, flash, debug, and measurement are included for development and production.

Note:

- Follow the ESP-IDF Programming Guide to set up the ESP-IDF environment.
- It is not recommended to install the ESP-IDF environment using the VSCode extension, as this may cause build failures for some examples that depend on the *esp_board_manager* component.

1.3 Hardware

ESP SoCs typically provide:

- Wi-Fi (2.4 GHz / 5 GHz dual band where applicable)

- Bluetooth 5.x (BLE / Mesh)
- High-performance multi-core CPUs (up to ~400 MHz)
- Ultra-low-power coprocessor and deep sleep
- **Rich peripherals:**
  - General-purpose: GPIO, UART, I2C, I2S, SPI, SDIO, USB OTG, etc.
  - Dedicated: LCD, camera, Ethernet, CAN, touch, LED PWM, temperature sensors, and more
- **Memory:**
  - Up to ~768 KB internal RAM
  - Optional external PSRAM
  - Optional external Flash
- **Security:**
  - Hardware crypto engine
  - Secure boot
  - Flash encryption
  - Digital signature

ESP SoCs use advanced process technology and offer leading RF performance, low power, and reliability for IoT, industrial, smart home, wearables, and similar applications.

Note:

- Refer to the ESP Product Selector for per-series details.
- Refer to ESP-Brookesia HAL Boards for supported boards.
- ESP-Brookesia requires a Flash capacity of at least 8 MB and a PSRAM capacity of at least 4 MB.

1.4 Obtaining and using components

Use the ESP Component Registry to add ESP-Brookesia components.

Example: add **brookesia_service_wifi**:

1. **Command line**
   From your project directory:

   ```
   idf.py add-dependency "espressif/brookesia_service_wifi"
   ```

2. **Manifest**
   Create or edit *idf_component.yml*:

   ```
   dependencies:
     espressif/brookesia_service_wifi: "*"
   ```

See ESP Registry Docs for more.

1.5 Using example projects

ESP-Brookesia ships multiple examples. Typical workflow:

1. Complete ESP-IDF setup first.
2. Select target chip or board (depends on peripherals):
   - **Chip only**:
     For `examples/service/wifi`, only Wi-Fi is required, so select a chip target (e.g. `esp32s3`):

     ```
     idf.py set-target <target>
     ```

   - **Board**:

For `examples/service/console`, audio peripherals matter, so select a board (e.g. `esp_vocat_board_v1_2`):

```
idf.py gen-bmgr-config -b <board>
idf.py set-target <target>
```

3. Optional configuration:

```
idf.py menuconfig
```

4. Build and flash:

```
idf.py build
idf.py -p <PORT> flash
```

5. Monitor serial output:

```
idf.py -p <PORT> monitor
```

More examples live under `examples/`; see each README for details.

Chapter 2

Utils Components

This section documents ESP-Brookesia utility components.

2.1 Library Utils

- Component registry: espressif/brookesia_lib_utils
- Public header: `#include "brookesia/lib_utils.hpp"`

2.1.1 Overview

brookesia_lib_utils is the general-purpose utility library for ESP-Brookesia: task scheduling, thread configuration, profiling, logging, state machines, plugins, and helpers.

2.1.2 Features

- Thread configuration: RAII-style thread settings (name, priority, stack, core affinity)
- Task scheduler: Boost.Asio–based async scheduling (immediate, delayed, periodic)
- Logging: ESP_LOG and printf-style output with formatting
- State machine and plugins: complex flows and modular extension
- Helpers: checks, function guards, describe/serialization
- Profilers: memory, thread, and time analysis

2.1.3 Core utilities

Thread Config

Public header: `#include "brookesia/lib_utils/thread_config.hpp"`

Overview *thread_config* centralizes thread runtime parameters and uses RAII guards to apply and restore settings in a scope.

Features

- *ThreadConfig* for name, priority, stack, core affinity
- Read defaults or current thread settings
- *ThreadConfigGuard* restores previous config on scope exit
- Convenience macros for configuration and queries

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/thread_config.hpp

Classes

class **ThreadConfigGuard**

RAII guard for thread configuration.

This class provides automatic restoration of thread configuration. When constructed, it applies the new configuration and saves the original. When destroyed, it restores the original configuration.

Public Functions

ThreadConfigGuard(const ThreadConfig &config)

Apply a new thread configuration and remember the previous one.

Parameters **config** –**[in]** Thread configuration to apply for the lifetime of this guard.

~ThreadConfigGuard()

Restore the thread configuration that was active before construction.

Macros

_BROOKESIA_THREAD_CONFIG_CONCAT(a, b)

BROOKESIA_THREAD_CONFIG_CONCAT(a, b)

BROOKESIA_THREAD_CONFIG_GUARD(...)

Apply a temporary thread configuration for the current scope.

This macro creates a `ThreadConfigGuard` instance that restores the previous configuration automatically when the surrounding scope exits.

```
{
    BROOKESIA_THREAD_CONFIG_GUARD({
         .stack_size = 10 * 1024,
    });
    boost::thread([&]() {
        // Thread will be created with 10KB stack size
    });
}  // Original configuration is restored here
```

Parameters

- **...** –Arguments forwarded to the `ThreadConfigGuard` constructor.

BROOKESIA_THREAD_GET_CURRENT_CONFIG()

Query the runtime configuration of the current task.

```
BROOKESIA_LOGI("Current task: %1%", BROOKESIA_THREAD_GET_CURRENT_CONFIG());
```

Note: This returns the actual runtime configuration of the current FreeRTOS task, which may differ from the applied pthread configuration if the task was created directly via FreeRTOS APIs rather than pthread APIs.

Returns `ThreadConfig` containing the current task's runtime configuration.

BROOKESIA_THREAD_GET_APPLIED_CONFIG()

Query the pthread configuration currently applied to newly created threads.

```
BROOKESIA_LOGI("Applied task: %1%", BROOKESIA_THREAD_GET_APPLIED_CONFIG());
```

Note: This returns the applied pthread configuration, which may differ from the actual runtime configuration of existing tasks. Use *BROOKESIA_THREAD_GET_CURRENT_CONFIG()* to get the actual runtime state of the current task.

Returns `ThreadConfig` containing the active pthread defaults. When no override was applied, the system default configuration is returned.

Task Scheduler

Public header: `#include "brookesia/lib_utils/task_scheduler.hpp"`

Overview *task_scheduler* is a Boost.Asio–based scheduler supporting immediate, delayed, and periodic tasks, with grouped serial/parallel execution.

Features

- One-shot, delayed, periodic, and batch scheduling
- Task groups and serial execution control
- Pause, resume, cancel, and wait for completion
- Statistics and pre/post callbacks

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/task_scheduler.hpp

Classes

class **TaskScheduler**

Asynchronous task scheduler built on top of `boost::asio::io_context`.

The scheduler supports immediate, delayed, and periodic tasks, optional task grouping, serial execution within groups, suspension and resumption of timer-driven tasks, and task lifecycle statistics.

Public Types

enum class **TaskType**

Kind of scheduled task.

Values:

enumerator **Immediate**

Task scheduled with *post()* or *dispatch()*.

enumerator **Delayed**

One-shot timer task scheduled with *post_delayed()*.

enumerator **Periodic**

Repeating timer task scheduled with *post_periodic()*.

enum class **TaskState**

Runtime state reported for a scheduled task.

Values:

enumerator **Running**

Task is active and eligible to execute.

enumerator **Suspended**

Task timer is suspended.

enumerator **Canceled**

Task was canceled before completion.

enumerator **Finished**

Task finished execution or is no longer tracked.

using **TaskId** = uint64_t

Identifier type assigned to each scheduled task.

using **OnceTask** = std::function<void()>

Callable type for one-shot tasks.

using **PeriodicTask** = std::function<bool()>

Callable type for periodic tasks.

Returning `false` stops future executions.

using **Group** = std::string

Task group name type.

using **Executor** = boost::asio::io_context::executor_type

Executor type exposed by the underlying `boost::asio::io_context`.

using **PreExecuteCallback** = std::function<void(const *Group*&, *TaskId*, *TaskType*)>

Callback invoked when a task is selected by io_context and about to execute.

Note: This callback is invoked after the task is selected by io_context and just before the actual task logic executes, guaranteeing that the task will execute (unless an exception occurs in the callback itself)

Param group Group name of the task
Param task_id ID of the task about to execute
Param task_type Type of the task

using **PostExecuteCallback** = std::function<void(const *Group*&, *TaskId*, *TaskType*, bool success)>

Callback invoked after a task completes execution.

Note: This callback is invoked after the task logic completes, regardless of success or failure, providing a hook for cleanup, logging, or statistics

---

Param group Group name of the completed task
Param task_id ID of the completed task
Param task_type Type of the task
Param success Whether the task executed successfully

Public Functions

TaskScheduler() = default

Construct an idle task scheduler.

~TaskScheduler()

Stop the scheduler and release its resources.

TaskScheduler(const *TaskScheduler*&) = delete

Copy construction is not supported.

TaskScheduler &**operator=**(const *TaskScheduler*&) = delete

Copy assignment is not supported.

TaskScheduler(*TaskScheduler*&&) = delete

Move construction is not supported.

TaskScheduler &**operator=**(*TaskScheduler*&&) = delete

Move assignment is not supported.

bool **start**(const *StartConfig* &config)

Start the scheduler with a custom worker configuration.

Parameters **config** –**[in]** Worker-thread and callback configuration.
Returns `true` on success, or `false` when startup fails.

inline bool **start**()

Start the scheduler with the default configuration.

Returns `true` on success, or `false` when startup fails.

void **stop**()

Stop the scheduler and cancel all pending tasks.

Running tasks are allowed to finish, worker threads are joined, and internal state is reset.

inline bool **is_running**() const

Check whether the scheduler is running.

Returns `true` when worker threads and the `io_context` are active, or `false` otherwise.

bool **configure_group**(const *Group* &group, const *GroupConfig* &config)

Configure execution behavior for a task group.

Parameters
- **group** –**[in]** Group name.
- **config** –**[in]** Group behavior configuration.

Returns `true` if the group was configured successfully, or `false` otherwise.

bool **dispatch**(*OnceTask* task, *TaskId* *id = nullptr, const *Group* &group = "")

Dispatch a task for immediate execution when possible.

When called from a scheduler worker thread, the task may run inline instead of being enqueued.

Parameters
- **task** –**[in]** Task to execute.
- **id** –**[out]** Optional pointer that receives the assigned task ID.
- **group** –**[in]** Optional task group name.

Returns `true` if the task was scheduled successfully, or `false` otherwise.

bool **post**(*OnceTask* task, *TaskId* *id = nullptr, const *Group* &group = "")

Post a task to the scheduler queue.

Parameters
- **task** –**[in]** Task to execute.
- **id** –**[out]** Optional pointer that receives the assigned task ID.
- **group** –**[in]** Optional task group name.

Returns `true` if the task was scheduled successfully, or `false` otherwise.

bool **post_delayed**(*OnceTask* task, int delay_ms, *TaskId* *id = nullptr, const *Group* &group = "")

Schedule a one-shot task to run after a delay.

Parameters
- **task** –**[in]** Task to execute.
- **delay_ms** –**[in]** Delay before execution, in milliseconds.
- **id** –**[out]** Optional pointer that receives the assigned task ID.
- **group** –**[in]** Optional task group name.

Returns `true` if the task was scheduled successfully, or `false` otherwise.

bool **post_periodic**(*PeriodicTask* task, int interval_ms, *TaskId* *id = nullptr, const *Group* &group = "")

Schedule a periodic task.

The task will be executed repeatedly at the specified interval. Return false from the task to stop the periodic execution.

Parameters
- **task** –**[in]** Periodic task to execute. Returning `false` stops future runs.
- **interval_ms** –**[in]** Interval between executions, in milliseconds.
- **id** –**[out]** Optional pointer that receives the assigned task ID.
- **group** –**[in]** Optional task group name.

Returns `true` if the task was scheduled successfully, or `false` otherwise.

bool **post_batch**(std::vector<*OnceTask*> tasks, std::vector<*TaskId*> *ids = nullptr, const *Group* &group = "")

Post multiple one-shot tasks as a batch.

Parameters
- **tasks** –**[in]** Vector of tasks to execute.
- **ids** –**[out]** Optional pointer that receives the assigned task IDs.
- **group** –**[in]** Optional task group name applied to every task.

Returns `true` if all tasks were scheduled successfully, or `false` otherwise.

void **cancel**(*TaskId* id)

Cancel a task by ID.

Parameters **id** –**[in]** Task ID to cancel.

void **cancel_group**(const *Group* &group)

Cancel every task in a group.

Parameters **group** –**[in]** Group name.

void **cancel_all**()

Cancel all tracked tasks.

bool **suspend**(*TaskId* id)

Suspend a delayed or periodic task.

Only delayed and periodic tasks can be suspended.

Parameters **id** –**[in]** Task ID to suspend.
Returns `true` if the task was suspended successfully, or `false` otherwise.

size_t **suspend_group**(const *Group* &group)

Suspend all suspendable tasks in a group.

Parameters **group** –**[in]** Group name.
Returns Number of tasks suspended.

size_t **suspend_all**()

Suspend all suspendable tasks.

Returns Number of tasks suspended.

bool **resume**(*TaskId* id)

Resume a suspended delayed or periodic task.

Parameters **id** –**[in]** Task ID to resume.
Returns `true` if the task was resumed successfully, or `false` otherwise.

size_t **resume_group**(const *Group* &group)

Resume all suspended tasks in a group.

Parameters **group** –**[in]** Group name.
Returns Number of tasks resumed.

size_t **resume_all**()

Resume all suspended tasks.

Returns Number of tasks resumed.

bool **wait**(*TaskId* id, int timeout_ms = -1)

Wait for a task to complete.

Parameters
- **id** –**[in]** Task ID to wait for.
- **timeout_ms** –**[in]** Timeout in milliseconds. `-1` waits indefinitely.

Returns `true` if the task completed within the timeout, or `false` otherwise.

bool **wait_group**(const *Group* &group, int timeout_ms = -1)

Wait for all tasks in a group to complete.

Parameters
- **group** –**[in]** Group name.
- **timeout_ms** –**[in]** Timeout in milliseconds. `-1` waits indefinitely.

Returns `true` if all tasks in the group completed within the timeout, or `false` otherwise.

bool **wait_all**(int timeout_ms = -1)

Wait for all tasks to complete.

Parameters **timeout_ms** –**[in]** Timeout in milliseconds. `-1` waits indefinitely.
Returns `true` if all tasks completed within the timeout, or `false` otherwise.

bool **restart_timer**(*TaskId* id)

Restart the timer for a delayed or periodic task.

This resets the timer countdown to its original interval. Useful for implementing debounce or watchdog-like behavior where you want to postpone execution.

Parameters **id** –**[in]** Task ID whose timer should restart.
Returns `true` if the timer restarted successfully, or `false` when the task is missing, has the wrong type, or is not running.

TaskType **get_type**(*TaskId* id) const

Query the type of a task.

Parameters **id** –**[in]** Task ID.
Returns Task type for the ID, or *`TaskType::Immediate`* when the task is unknown.

TaskState **get_state**(*TaskId* id) const

Query the state of a task.

Parameters **id** –**[in]** Task ID.
Returns Task state for the ID, or *`TaskState::Finished`* when the task is unknown.

Group **get_group**(*TaskId* id) const

Query the group of a task.

Parameters **id** –**[in]** Task ID.
Returns Group name, or an empty string when the task is unknown or ungrouped.

size_t **get_group_task_count**(const *Group* &group) const

Query the number of tracked tasks in a group.

Parameters **group** –**[in]** Group name.
Returns Number of tasks currently associated with the group.

std::vector<*Group*> **get_active_groups**() const

Get all groups that currently contain tasks.

Returns Vector of active group names.

Statistics **get_statistics**() const

Get execution statistics accumulated by the scheduler.

Returns *`Statistics`* structure with task counters.

inline std::shared_ptr<*Executor*> **get_executor**()

Get a shared executor handle for the underlying `io_context`.

Returns Shared pointer to the executor, or `nullptr` when the scheduler is not running.

inline size_t **get_worker_count**() const

Get the number of worker threads.

Returns Number of worker threads currently owned by the scheduler.

void **reset_statistics**()

Reset all accumulated statistics counters to zero.

struct **GroupConfig**

Configuration for a task group.

Public Members

bool **enable_serial_execution** = false

Serialize all tasks in this group through a strand when set to `true`.

Periodic tasks skip overlapping executions for the same task instance, while one-shot tasks are queued and executed sequentially.

Group **parent_group** = ""

Optional parent group name whose execution context should be reused.

When the parent group is serial, this group also runs serially through the parent strand.

PreExecuteCallback **pre_execute_callback** = nullptr

Optional group pre-execute callback applied to all tasks in the group.

PostExecuteCallback **post_execute_callback** = nullptr

Optional group post-execute callback applied to all tasks in the group.

struct **StartConfig**

Startup configuration for the scheduler worker threads.

Public Members

std::vector< ThreadConfig > worker_configs = { ThreadConfig{.name = "Worker",.stack_size = 6 * 1024,}}

Worker thread configurations used when the scheduler starts.

The default configuration creates a single worker named `Worker` with a 6 KiB stack.

size_t **worker_poll_interval_ms** = 10

Worker polling interval in milliseconds.

PreExecuteCallback **pre_execute_callback** = nullptr

Optional global pre-execute callback applied to every task.

The callback fires just before any task executes, regardless of group membership.

PostExecuteCallback **post_execute_callback** = nullptr

Optional global post-execute callback applied to every task.

The callback fires after any task completes, regardless of group membership.

struct **Statistics**

Aggregate counters for task execution.

Public Members

size_t **total_tasks** = {0}

Total number of tasks ever scheduled.

size_t **completed_tasks** = {0}

Number of tasks that completed successfully.

size_t **failed_tasks** = {0}

Number of tasks that finished unsuccessfully.

size_t **canceled_tasks** = {0}

Number of tasks canceled before completion.

size_t **suspended_tasks** = {0}

Number of tasks currently suspended.

State Base

Public header: `#include "brookesia/lib_utils/state_base.hpp"`

Overview *state_base* defines the base state abstraction for state machines with enter, exit, and update callbacks, plus timeouts and update intervals.

Features

- Standard lifecycle: *on_enter* / *on_exit* / *on_update*
- Timeout actions and update intervals
- Easy to subclass for custom states

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/state_base.hpp

Classes

class **StateBase**

Base class for state machine states.

Provides lifecycle hooks (on_enter, on_exit, on_update) and configuration for timeout and periodic update intervals.

Public Functions

inline explicit **StateBase**(const std::string &name)

Constructor.

Parameters **name** –State name.

virtual **~StateBase**() = default

Virtual destructor.

inline virtual bool **on_enter**(const std::string &from_state = "", const std::string &action = "")

Hook invoked before the state becomes active.

Parameters
- **from_state** –Name of the previous state, or an empty string for the initial state.
- **action** –Transition action name, or an empty string when not specified.

Returns `true` to allow the transition, or `false` to reject entry.

inline virtual bool **on_exit**(const std::string &to_state = "", const std::string &action = "")

Hook invoked before the state is left.

Parameters
- **to_state** –Name of the next state, or an empty string when unspecified.
- **action** –Transition action name, or an empty string when not specified.

Returns `true` to allow the transition, or `false` to reject exit.

inline virtual void **on_update**()

Hook invoked periodically while the state remains active.

Note: This callback is scheduled only when *`set_update_interval()`* configures a non-zero interval.

---

inline void **set_timeout**(uint32_t ms, const std::string &action)

Configure an automatic timeout transition for this state.

Parameters
- **ms** –Timeout duration in milliseconds. A value of `0` disables the timeout.
- **action** –Action name triggered when the timeout expires.

inline void **set_update_interval**(uint32_t interval_ms)

Configure the periodic update interval for this state.

Parameters **interval_ms** –Update period in milliseconds. A value of `0` disables *`on_update()`* scheduling.

inline const std::string &**get_name**() const

Get the state name.

Returns State name.

inline uint32_t **get_timeout_ms**() const

Get the configured timeout duration.

Returns Timeout duration in milliseconds, or `0` when no timeout is configured.

inline const std::string &**get_timeout_action**() const

Get the action triggered when the timeout expires.

Returns Configured timeout action name.

inline uint32_t **get_update_interval**() const

Get the configured periodic update interval.

Returns Update interval in milliseconds, or `0` when periodic updates are disabled.

State Machine

Public header: `#include "brookesia/lib_utils/state_machine.hpp"`

Overview *state_machine* implements registration, actions, transition callbacks, and runtime control for multi-state workflows.

Features

- Add states and transition rules
- Initial state and forced switches
- Transition callbacks and runtime queries
- Optional integration with *TaskScheduler* for async work

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/state_machine.hpp

Classes

class **StateMachine**

Thread-safe Finite State Machine implementation.

Manages states, transitions, and state lifecycle with support for:

- State entry/exit guards (on_enter/on_exit)
- Periodic state updates (on_update)
- State timeouts with automatic action triggering
- Asynchronous state transitions via action queue
- Serial execution guarantee (no concurrent state transitions)
- Transition rollback on entry failure

Note: All state transitions are executed serially through the task scheduler's group mechanism, ensuring thread-safe operations even when actions are triggered from multiple threads. State callbacks (on_enter, on_exit, on_update) are executed without holding internal locks, allowing them to safely trigger new actions or perform blocking operations.

Public Types

using **StatePtr** = std::shared_ptr<*StateBase*>

Shared pointer type used for registered states.

using **TransitionFinishCallback** = std::function<void(const std::string &from, const std::string &action, const std::string &to)>

Callback type invoked after a transition completes successfully.

Public Functions

StateMachine() = default

Constructor.

~StateMachine()

Stop the state machine and release owned resources.

bool **add_state**(*StatePtr* state)

Register a named state object.

Parameters **state** –Shared pointer to the state implementation.
Returns `true` if the state was added, or `false` when a state with the same name already exists.

bool **add_transition**(const std::string &from, const std::string &action, const std::string &to)

Register a transition between two states.

Parameters
- **from** –Source state name.
- **action** –Action name that triggers the transition.
- **to** –Target state name.

Returns `true` if the transition was added, or `false` when the same `(from, action)` mapping already exists.

bool **start**(*Config* config)

Start the state machine and enter the initial state.

Note: If the state machine is already running, this returns true immediately. If the scheduler is not running, this will attempt to start it. The task group is automatically configured for serial execution to ensure thread-safe transitions.

Parameters **config** –Configuration for the state machine.
Returns `true` on success, or `false` if the initial state cannot be entered or startup fails.

void **stop**()

Stop the state machine and wait for pending scheduled work to finish.

Note: This method is thread-safe and can be called multiple times safely. It will wait for all pending state tasks to complete before returning.

---

bool **trigger_action**(const std::string &action, bool use_dispatch = false)

Queue or dispatch a transition action.

Note: The actual transition happens asynchronously in the task scheduler's serial queue. This ensures thread-safe state transitions even when called from multiple threads.

Parameters
- **action** –Transition action name.
- **use_dispatch** –Set to `true` to call *`TaskScheduler::dispatch()`* instead of `post()`.

Returns `true` if the action was scheduled successfully, or `false` otherwise.

bool **wait_all_transitions**(uint32_t timeout_ms)

Wait until no transitions are queued or running.

Parameters **timeout_ms** – Timeout in milliseconds. The special value `static_cast<uint32_t>(-1)` waits indefinitely.

Returns `true` if all transitions completed before the timeout, or `false` otherwise.

bool **force_transition_to**(const std::string &target_state)

Cancel scheduled state tasks and overwrite the current state name immediately.

Note: This bypasses `on_exit()`, `on_enter()`, timeout scheduling, periodic updates, and transition callbacks.

Parameters **target_state** – State name assigned as the new current state.

Returns `true` once the internal state name has been updated.

inline void **register_transition_finish_callback**(*TransitionFinishCallback* callback)

Register a callback invoked after a successful transition.

Note: The callback is invoked with (from_state, action, to_state) parameters. The callback will be executed without holding internal locks, so it's safe to perform any operations including triggering new actions.

Parameters **callback** – Callback invoked with `(from_state, action, to_state)`.

inline bool **is_running**() const

Check whether the state machine has been started.

Note: This method is thread-safe.

Returns `true` when the machine owns a scheduler and is running, or `false` otherwise.

inline bool **has_transition_running**() const

Check whether any transitions are queued or currently executing.

Note: This method is thread-safe.

Returns `true` when one or more transitions are pending or in progress, or `false` otherwise.

inline bool **has_state_updating**() const

Check whether the current state has an active periodic update task.

Note: This method is thread-safe.

Returns `true` when a periodic update task is installed, or `false` otherwise.

inline std::string **get_current_state**() const

Get the name of the current state.

Note: This method is thread-safe and returns a copy of the state name.

Returns Current state name, or an empty string when the machine has not started.

inline *StatePtr* **get_state_ptr**(const std::string &name) const

Look up a registered state by name.

Note: This method is thread-safe.

Parameters **name** – State name to look up.
Returns Shared pointer to the registered state, or `nullptr` if not found.

Public Static Attributes

static constexpr const char ***DEFAULT_TASK_GROUP_NAME** = "state_machine"

Default scheduler group name used by the state machine.

struct **Config**

Configuration for the state machine.

Public Members

std::shared_ptr<*TaskScheduler*> **task_scheduler**

Task scheduler used to serialize transitions, timeouts, and periodic updates.

std::string **task_group_name** = *DEFAULT_TASK_GROUP_NAME*

Task group name used for serialized state-machine tasks.

TaskScheduler::GroupConfig task_group_config = {.
enable_serial_execution = true,}

Task group configuration.

std::string **initial_state**

Initial state name.

Plugin

Public header: `#include "brookesia/lib_utils/plugin.hpp"`

Overview *plugin* provides a generic plugin registry and instance management: register, discover, and create plugins by name.

Features

- Template registration and lookup by name
- Lazy instantiation and singleton registration
- Enumerate, release, and cleanup
- Macros for registration symbols and link visibility

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/plugin.hpp

Classes

template<typename **T**>

class **PluginRegistry**

Thread-safe registry for named plugin factories and cached instances.

Template Parameters **T** – Base class type exposed by the registry.

Public Static Functions

static inline std::shared_ptr<*T*> **get_instance**(const std::string &name)

Get a plugin instance by name.

Returns the cached instance when available, otherwise creates it through the registered factory.

Note: The returned shared_ptr shares ownership with the Registry. The instance remains valid as long as either the Registry or any returned shared_ptr holds a reference, ensuring thread-safe access even if *release_instance()* or *remove_plugin()* is called.

Parameters **name** – **[in]** Registered plugin name.
Returns Shared pointer to the plugin instance, or `nullptr` when the plugin is not registered or the factory is empty.

static inline std::map<std::string, std::shared_ptr<*T*>> **get_all_instances**()

Get instances for all registered plugins.

Missing cached instances are created on demand before they are returned.

Note: Each returned shared_ptr shares ownership with the Registry. The instances remain valid as long as either the Registry or any returned shared_ptr holds a reference, ensuring thread-safe access.

Returns Map from plugin name to shared plugin instance.

static inline size_t **get_plugin_count**()

Get the number of registered plugin names.

Returns Total number of registered plugins.

static inline bool **has_plugin**(const std::string &name)

Check whether a plugin name is registered.

Parameters **name** –**[in]** Plugin name to test.
Returns `true` when the plugin exists in the registry, or `false` otherwise.

static inline void **release_instance**(const std::string &name)

Release the cached instance for a registered plugin.

This keeps the factory registration intact and only drops the registry-held shared pointer.

Parameters **name** –**[in]** Plugin name.

static inline void **release_all_instances**()

Release all cached plugin instances without removing registrations.

static inline void **remove_plugin**(const std::string &name)

Remove a plugin registration and its cached instance.

Parameters **name** –**[in]** Plugin name to remove.

static inline void **remove_all_plugins**()

Remove all registered plugins and cached instances.

template<typename **PluginType**>
static inline void **register_plugin**(const std::string &name, FactoryFunc factory)

Register a plugin factory under a unique name.

Template Parameters **PluginType** –Concrete plugin type being registered.
Parameters

- **name** –**[in]** Plugin name. Existing registrations with the same name are kept unchanged.
- **factory** –**[in]** Factory used to lazily create instances.

Macros

_BROOKESIA_PLUGIN_CONCAT(a, b)

BROOKESIA_PLUGIN_CONCAT(a, b)

BROOKESIA_PLUGIN_CREATE_SYMBOL(symbol_name, static_var_name)

Create a linker-visible symbol that keeps a registrar object alive.

Note: The function is defined outside any namespace to ensure proper linking. The static variable reference ensures the registrar instance is not optimized away.

BROOKESIA_PLUGIN_REGISTER_WITH_CONSTRUCTOR(BaseType, PluginType, name, creator, symbol_name)

Register a plugin using a custom creator expression.

Note: Automatically creates a fixed symbol name based on PluginType for linker -u option

Parameters

- **BaseType** –Registry base type.
- **PluginType** –Concrete plugin type to register.
- **name** –Plugin name used by `PluginRegistry`.
- **creator** –Creator expression returning `shared_ptr` or `unique_ptr` compatible with the registry.
- **symbol_name** –Linker symbol exported for `-u`.

BROOKESIA_PLUGIN_REGISTER(BaseType, PluginType, name, ...)

Register a plugin constructed with `std::make_shared`.

```
// Example usage:
BROOKESIA_PLUGIN_REGISTER(BaseService, MyPlugin, "my_plugin");
BROOKESIA_PLUGIN_REGISTER(BaseService, MyPlugin, "my_plugin", arg1, arg2);
```

Parameters

- **BaseType** –Registry base type.
- **PluginType** –Concrete plugin type to register.
- **name** –Plugin name used by `PluginRegistry`.
- `...` –Optional constructor arguments forwarded to `PluginType`.

BROOKESIA_PLUGIN_REGISTER_SINGLETON(BaseType, PluginType, name, instance_expr)

Register a singleton object in the plugin registry.

This macro allows registering a singleton instance to the plugin registry. It uses a custom no-op deleter to prevent the shared_ptr from destroying the singleton. Automatically generates a fixed symbol name based on PluginType for linker -u option.

```
// Example usage:
BROOKESIA_PLUGIN_REGISTER_SINGLETON(
    BaseService,
    MySingleton,
    "my_singleton",
    MySingleton::get_instance()
);
// Creates symbol: _MySingleton_symbol_<line_number>
// Use: target_link_libraries(${COMPONENT_LIB} INTERFACE "-u _MySingleton_
↪symbol_<line_number>")
```

Parameters

- **BaseType** –Registry base type.
- **PluginType** –Singleton type to register.
- **name** –Plugin name used by `PluginRegistry`.
- **instance_expr** –Expression that yields a singleton instance reference, for example `Type::get_instance()`.

BROOKESIA_PLUGIN_REGISTER_WITH_SYMBOL(BaseType, PluginType, name, symbol_name, ...)

Register a plugin constructed with `std::make_shared` and a custom linker symbol.

```
// Example usage:
BROOKESIA_PLUGIN_REGISTER_WITH_SYMBOL(BaseService, MyPlugin, "my_plugin",
↪"custom_symbol_name");
BROOKESIA_PLUGIN_REGISTER_WITH_SYMBOL(BaseService, MyPlugin, "my_plugin",
↪"custom_symbol_name", arg1, arg2);
// Use: target_link_libraries(${COMPONENT_LIB} INTERFACE "-u custom_symbol_name
↪")
```

Parameters

- **BaseType** –Registry base type.
- **PluginType** –Concrete plugin type to register.
- **name** –Plugin name used by `PluginRegistry`.
- **symbol_name** –Custom linker symbol exported for `-u`.

- `...` –Optional constructor arguments forwarded to `PluginType`.

BROOKESIA_PLUGIN_REGISTER_SINGLETON_WITH_SYMBOL(BaseType, PluginType, name, instance_expr, symbol_name)

Register a singleton object with a custom linker symbol.

This macro allows registering a singleton instance to the plugin registry. It uses a custom no-op deleter to prevent the shared_ptr from destroying the singleton. Uses the provided custom symbol name for linker -u option.

```
// Example usage:
BROOKESIA_PLUGIN_REGISTER_SINGLETON_WITH_SYMBOL(
    BaseService,
    MySingleton,
    "my_singleton",
    MySingleton::get_instance(),
    "custom_symbol_name"
);
// Use: target_link_libraries(${COMPONENT_LIB} INTERFACE "-u custom_symbol_name
↪")
```

Parameters

- **BaseType** –Registry base type.
- **PluginType** –Singleton type to register.
- **name** –Plugin name used by `PluginRegistry`.
- **instance_expr** –Expression that yields a singleton instance reference.
- **symbol_name** –Custom linker symbol exported for `-u`.

2.1.4 Helpers

Logging

Public header: `#include "brookesia/lib_utils/log.hpp"`

Overview *log* provides a unified logging API with formatting, switching between ESP_LOG and standard output, and *std::source_location* context.

Features

- Levels: Trace / Debug / Info / Warn / Error
- Unified formatting compatible with *boost::format*-style placeholders
- Automatic function/file context
- Trace Guard for enter/exit logging

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/log.hpp

Classes

class **Log**

Logging backend facade used by the public logging macros.

Public Functions

inline void **print**(int level, const std::source_location &loc, const char *tag, const char *format)

Print a preformatted message.

Parameters
- **level** –*Log* level defined by `BROOKESIA_UTILS_LOG_LEVEL_*`.
- **loc** –Source location used for metadata extraction.
- **tag** –*Log* tag string.
- **format** –Message string passed through without argument formatting.

template<typename ...**Args**>
inline void **print**(int level, const std::source_location &loc, const char *tag, const char *format, *Args*&&... args)

Format a message and print it.

Template Parameters **Args** –Format argument types.
Parameters
- **level** –*Log* level defined by `BROOKESIA_UTILS_LOG_LEVEL_*`.
- **loc** –Source location used for metadata extraction.
- **tag** –*Log* tag string.
- **format** –Boost-format-style message format.
- **args** –Format arguments forwarded through the type-erased formatter.

Public Static Functions

static inline *Log* &**getInstance**()

Get the singleton logging facade.

Returns Reference to the shared *Log* instance.

static void **write**(int level, const std::source_location &loc, const char *tag, const std::string &message)

Emit a fully formatted message through the selected backend.

Parameters
- **level** –*Log* level defined by `BROOKESIA_UTILS_LOG_LEVEL_*`.
- **loc** –Source location used for metadata extraction.
- **tag** –*Log* tag string.
- **message** –Final message body.

static std::string **format_message**(const char *format, std::initializer_list<FormatArg> args)

Format a message using `boost::format`-style placeholders.

Parameters
- **format** –Format string.
- **args** –Type-erased format arguments.

Returns Formatted message string.

static std::string_view **extract_function_name**(const char *func_name)

Extract the display-friendly function name from a source location string.

Parameters **func_name** –Raw function signature string.
Returns Trimmed function name view.

static std::string_view **extract_file_name**(const char *file_path)

Extract the leaf file name from a source path.

Parameters **file_path** –Raw source file path.
Returns File name view without directory components.

template<bool **Enabled**>

class **LogTraceGuard**

RAII helper that logs function entry and exit at trace level.

Template Parameters **Enabled** –Compile-time flag that removes all work when `false`.

Public Functions

inline **LogTraceGuard**(const void *this_ptr = nullptr, const std::source_location &loc = std::source_location::current(), const char *tag = esp_brookesia::lib_utils::TAG)

Construct a trace guard for the current scope.

Parameters
- **this_ptr** –Optional object pointer logged for member functions.
- **loc** –Source location captured for the scope.
- **tag** –*Log* tag string.

inline **~LogTraceGuard**()

Log scope exit when trace logging is enabled.

Macros

_BROOKESIA_LOG_FORMAT_THREAD_NAME

Helper macros to assemble format string and arguments based on enabled macros These macros use string literal concatenation (adjacent string literals are automatically concatenated)

_BROOKESIA_LOG_FORMAT_FILE_LINE

_BROOKESIA_LOG_FORMAT_FUNCTION

_BROOKESIA_LOG_FORMAT_MESSAGE

_BROOKESIA_LOG_FORMAT_STRING

_BROOKESIA_LOG_ARGS_THREAD_NAME(thread_name)

_BROOKESIA_LOG_ARGS_FILE_LINE(file_name, line)

_BROOKESIA_LOG_ARGS_FUNCTION(func_name)

_BROOKESIA_LOG_ARGS_MESSAGE(format_str)

_BROOKESIA_LOG_ARGS(thread_name, file_name, line, func_name, format_str)

BROOKESIA_LOGT_IMPL(tag, format, ...)

Macros to simplify logging calls with a fixed tag

BROOKESIA_LOGD_IMPL(tag, format, ...)

BROOKESIA_LOGI_IMPL(tag, format, ...)

BROOKESIA_LOGW_IMPL(tag, format, ...)

BROOKESIA_LOGE_IMPL(tag, format, ...)

BROOKESIA_LOG_DISABLE_DEBUG_TRACE

Per-file switch that disables `BROOKESIA_LOGT`, `BROOKESIA_LOGD`, and trace-guard helpers.

Users can define this macro before including this header to override the default behavior.

```
#define BROOKESIA_LOG_DISABLE_DEBUG_TRACE 1  // Disable all debug & trace␣
↪features for this file
#include "brookesia/lib_utils/log.hpp"
```

BROOKESIA_LOGT(format, ...)

Compile-time log level filtering macros.

Calls below the configured log level expand to no-ops.

Emit a trace-level log message for the current translation unit tag.

BROOKESIA_LOGD(format, ...)

Emit a debug-level log message for the current translation unit tag.

BROOKESIA_LOGI(format, ...)

Emit an info-level log message for the current translation unit tag.

BROOKESIA_LOGW(format, ...)

Emit a warning-level log message for the current translation unit tag.

BROOKESIA_LOGE(format, ...)

Emit an error-level log message for the current translation unit tag.

_BROOKESIA_LOG_TRACE_FORMAT_ENTER_WITH_PTR

_BROOKESIA_LOG_TRACE_FORMAT_ENTER

_BROOKESIA_LOG_TRACE_FORMAT_EXIT_WITH_PTR

_BROOKESIA_LOG_TRACE_FORMAT_EXIT

_BROOKESIA_LOG_TRACE_FORMAT_ENTER_WITH_PTR_STRING

_BROOKESIA_LOG_TRACE_FORMAT_ENTER_STRING

_BROOKESIA_LOG_TRACE_FORMAT_EXIT_WITH_PTR_STRING

_BROOKESIA_LOG_TRACE_FORMAT_EXIT_STRING

_BROOKESIA_LOG_TRACE_ARGS_WITH_PTR(thread_name, file_name, line, func_name, this_ptr)

_BROOKESIA_LOG_CONCAT(a, b)

Create a scope guard that logs entry and exit for the current function.

BROOKESIA_LOG_CONCAT(a, b)

BROOKESIA_LOG_TRACE_GUARD()

BROOKESIA_LOG_TRACE_GUARD_WITH_THIS()

Create a scope guard that logs entry and exit and includes `this`.

Check

Public header: `#include "brookesia/lib_utils/check.hpp"`

Overview *check* provides unified check macros for invalid parameters, failed expressions, exceptions, error codes, and out-of-range values.

Features

- Covers common cases: `nullptr`, booleans, *std::exception*, *esp_err_t*, ranges
- Multiple failure modes: code blocks, *return*, *exit*, *goto*
- Integrates with logging for context
- Configurable failure policy (no-op, log, assert)

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/check.hpp

Macros

BROOKESIA_CHECK_NULL_EXECUTE(ptr, process_code, ...)

Execute a fallback code block when a pointer is null.

Parameters

- **ptr** –Pointer expression to test.
- **process_code** –Code block executed when `ptr` is `nullptr`.
- `...` –Compatibility arguments kept for legacy call sites and ignored in this mode.

BROOKESIA_CHECK_FALSE_EXECUTE(value, process_code, ...)

Execute a fallback code block when an expression evaluates to `false`.

Parameters

- **value** –Expression to test.
- **process_code** –Code block executed when `value` converts to `false`.
- `...` –Compatibility arguments kept for legacy call sites and ignored in this mode.

BROOKESIA_CHECK_EXCEPTION_EXECUTE(expression, process_code, ...)

Execute a fallback code block when an expression throws `std::exception`.

Parameters

- **expression** –Expression to evaluate inside a `try` block.
- **process_code** –Code block executed when `expression` throws `std::exception`.
- `...` –Compatibility arguments kept for legacy call sites and ignored in this mode.

BROOKESIA_CHECK_ESP_ERR_EXECUTE(esp_result, process_code, ...)

Execute a fallback code block when an ESP-IDF error code is not `ESP_OK`.

Parameters

- **esp_result** –Expression that yields an ESP-IDF error code.
- **process_code** –Code block executed when `esp_result` is not `ESP_OK`.
- `...` –Compatibility arguments kept for legacy call sites and ignored in this mode.

BROOKESIA_CHECK_OUT_RANGE_EXECUTE(value, min, max, process_code, ...)

Execute a fallback code block when a value is outside the inclusive range [`min`, `max`].

Parameters

- **value** –Expression to test.
- **min** –Inclusive lower bound.
- **max** –Inclusive upper bound.
- **process_code** –Code block executed when `value` is outside the range.
- `...` –Compatibility arguments kept for legacy call sites and ignored in this mode.

BROOKESIA_CHECK_NULL_RETURN(value, ret, fmt, ...)

Return a value when a pointer is null.

Parameters

- **value** –Pointer expression to test.
- **ret** –Value returned when `value` is `nullptr`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_NULL_EXIT(value, fmt, ...)

Return from the current `void` function when a pointer is null.

Parameters

- **value** –Pointer expression to test.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_NULL_GOTO(value, goto_tag, fmt, ...)

Jump to a label when a pointer is null.

Parameters

- **value** –Pointer expression to test.
- **goto_tag** –Label jumped to when `value` is `nullptr`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_FALSE_RETURN(value, ret, fmt, ...)

Return a value when an expression evaluates to `false`.

Parameters

- **value** –Expression to test.
- **ret** –Value returned when `value` converts to `false`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_FALSE_EXIT(value, fmt, ...)

Return from the current `void` function when an expression evaluates to `false`.

Parameters

- **value** –Expression to test.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_FALSE_GOTO(value, goto_tag, fmt, ...)

Jump to a label when an expression evaluates to `false`.

Parameters

- **value** –Expression to test.
- **goto_tag** –Label jumped to when `value` converts to `false`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_ESP_ERR_RETURN(value, ret, fmt, ...)

Return a value when an ESP-IDF error code is not `ESP_OK`.

Parameters

- **value** –Expression that yields an ESP-IDF error code.
- **ret** –Value returned when `value` is not `ESP_OK`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_ESP_ERR_EXIT(value, fmt, ...)

Return from the current `void` function when an ESP-IDF error code is not `ESP_OK`.

Parameters

- **value** –Expression that yields an ESP-IDF error code.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_ESP_ERR_GOTO(value, goto_tag, fmt, ...)

Jump to a label when an ESP-IDF error code is not `ESP_OK`.

Parameters

- **value** –Expression that yields an ESP-IDF error code.
- **goto_tag** –Label jumped to when `value` is not `ESP_OK`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_EXCEPTION_RETURN(expression, ret, fmt, ...)

Return a value when an expression throws `std::exception`.

Parameters

- **expression** –Expression to evaluate inside a `try` block.
- **ret** –Value returned when `expression` throws `std::exception`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_EXCEPTION_EXIT(expression, fmt, ...)

Return from the current `void` function when an expression throws `std::exception`.

Parameters

- **expression** –Expression to evaluate inside a `try` block.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_EXCEPTION_GOTO(expression, goto_tag, fmt, ...)

Jump to a label when an expression throws `std::exception`.

Parameters

- **expression** –Expression to evaluate inside a `try` block.
- **goto_tag** –Label jumped to when `expression` throws `std::exception`.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_OUT_RANGE(value, min, max, fmt, ...)

Log when a value is outside the inclusive range [`min`, `max`].

Parameters

- **value** –Expression to test.
- **min** –Inclusive lower bound.
- **max** –Inclusive upper bound.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- `...` –Optional format arguments for `fmt`.

BROOKESIA_CHECK_OUT_RANGE_RETURN(value, min, max, ret, fmt, ...)

Return a value when a value is outside the inclusive range [`min`, `max`].

Parameters

- **value** –Expression to test.
- **min** –Inclusive lower bound.
- **max** –Inclusive upper bound.
- **ret** –Value returned when `value` is outside the range.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- **...** –Optional format arguments for `fmt`.

BROOKESIA_CHECK_OUT_RANGE_EXIT(value, min, max, fmt, ...)

Return from the current `void` function when a value is outside the inclusive range [`min`, `max`].

Parameters

- **value** –Expression to test.
- **min** –Inclusive lower bound.
- **max** –Inclusive upper bound.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- **...** –Optional format arguments for `fmt`.

BROOKESIA_CHECK_OUT_RANGE_GOTO(value, min, max, goto_tag, fmt, ...)

Jump to a label when a value is outside the inclusive range [`min`, `max`].

Parameters

- **value** –Expression to test.
- **min** –Inclusive lower bound.
- **max** –Inclusive upper bound.
- **goto_tag** –Label jumped to when `value` is outside the range.
- **fmt** –User log format string forwarded to `BROOKESIA_LOGE`.
- **...** –Optional format arguments for `fmt`.

Function Guard

Public header: `#include "brookesia/lib_utils/function_guard.hpp"`

Overview *FunctionGuard* is an RAII guard that runs cleanup at scope exit to simplify resource release and rollback.

Features

- Automatic callback on scope exit; optional *release* to cancel
- Move semantics for transferring guards
- Catches *std::exception* during cleanup to avoid breaking teardown

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/function_guard.hpp

Classes

template<typename **T**, typename ...**Args**>

class **FunctionGuard**

RAII helper that invokes a callable on destruction unless released.

Note: The destructor suppresses `boost::thread_interrupted` and prints other `std::exception` failures to `stdout`, so cleanup code should still be kept lightweight.

Template Parameters

- **T** –Callable type stored by the guard.
- **Args** –Argument types forwarded to the callable when the guard is destroyed.

Public Functions

inline **FunctionGuard**(*T* func, *Args*&&... args)

Construct a guard for a deferred callable invocation.

Parameters

- **func** –Callable executed in the destructor unless *`release()`* is called.
- **args** –Arguments stored and forwarded to `func` during destruction.

inline void **release**()

Disable the deferred invocation.

After calling this method, destroying the guard no longer executes the stored callable.

Describe Helpers

Public header: `#include "brookesia/lib_utils/describe_helpers.hpp"`

Overview *describe_helpers* uses Boost.Describe and Boost.JSON for object description plus serialization and deserialization, reducing boilerplate for structs, enums, and complex types.

Features

- Reflection for enums and struct members
- JSON encode/decode for strings, numbers, bools, containers, optionals, etc.
- Handles *variant*, *optional*, *map*, *vector*, and related composites
- Debug-friendly description output

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/describe_helpers.hpp

Classes

class **DescribeFormatManager**

Singleton that stores the global default `DescribeOutputFormat`.

Public Functions

inline void **set_format**(const DescribeOutputFormat &fmt)

Set the process-wide default output format.

Parameters **fmt** –New global format.

inline const DescribeOutputFormat &**get_format**() const

Get the current process-wide default output format.

Returns Reference to the active global format.

inline void **reset_to_default**()

Restore the built-in default output format.

Public Static Functions

static inline *DescribeFormatManager* &**instance**()

Get the singleton instance.

Returns Reference to the global format manager.

Macros

BROOKESIA_DESCRIBE_STRUCT(C, Bases, Members)

Register a struct with Boost.Describe reflection.

BROOKESIA_DESCRIBE_ENUM(C, ...)

Register an enum with Boost.Describe reflection.

BROOKESIA_DESCRIBE_ENUM_TO_STR(value)

Convert a described enum value to a string.

BROOKESIA_DESCRIBE_ENUM_TO_NUM(value)

Convert an enum value to its underlying integer value.

BROOKESIA_DESCRIBE_NUM_TO_ENUM(number, ret_value)

Convert an integer value to a described enum value.

BROOKESIA_DESCRIBE_STR_TO_ENUM(str, ret_value)

Convert an enumerator name to a described enum value.

BROOKESIA_DESCRIBE_TO_JSON(value)

Convert a supported value to `boost::json::value`.

BROOKESIA_DESCRIBE_FROM_JSON(json_value, ret_value)

Convert a `boost::json::value` to a supported C++ value.

BROOKESIA_DESCRIBE_JSON_SERIALIZE(value)

Serialize a supported value to JSON text.

BROOKESIA_DESCRIBE_JSON_DESERIALIZE(str, ret_value)

Deserialize JSON text into a supported C++ value.

BROOKESIA_DESCRIBE_FORMAT_VERBOSE

Verbose multi-line formatting preset.

BROOKESIA_DESCRIBE_FORMAT_JSON

JSON-like formatting preset.

BROOKESIA_DESCRIBE_FORMAT_COMPACT

Compact single-line formatting preset.

BROOKESIA_DESCRIBE_FORMAT_DEFAULT

Default human-readable formatting preset.

BROOKESIA_DESCRIBE_FORMAT_PYTHON

Python-dict-inspired formatting preset.

BROOKESIA_DESCRIBE_FORMAT_CPP

C++ designated-initializer-inspired formatting preset.

BROOKESIA_DESCRIBE_SET_GLOBAL_FORMAT(fmt)

Set the global default string formatting preset.

BROOKESIA_DESCRIBE_GET_GLOBAL_FORMAT()

Get the global default string formatting preset.

BROOKESIA_DESCRIBE_RESET_GLOBAL_FORMAT()

Reset the global default string formatting preset.

BROOKESIA_DESCRIBE_TO_STR(value)

Convert a supported value to a string with the global default format.

BROOKESIA_DESCRIBE_TO_STR_WITH_FMT(value, fmt)

Convert a supported value to a string with an explicit format preset.

2.1.5 Debug tools

Memory Profiler

Public header: `#include "brookesia/lib_utils/memory_profiler.hpp"`

Overview *memory_profiler* samples heap usage, tracks peaks, and supports threshold alerts for runtime memory monitoring.

Features

- Current heap, historical peak, and statistics
- Periodic profiling and manual snapshots
- Threshold callbacks when limits are exceeded
- Optional integration with *TaskScheduler* for periodic sampling

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/memory_profiler.hpp

Classes

class **MemoryProfiler**

Heap memory profiler for internal RAM and PSRAM usage.

This class provides a C++ interface to monitor ESP32 heap memory usage, including internal SRAM and external PSRAM. It integrates naturally with *TaskScheduler* for periodic profiling.

Public Types

enum class **ThresholdType**

Metric used when registering threshold callbacks.

Values:

enumerator **TotalFree**

Total free memory in bytes.

enumerator **InternalFree**

Internal SRAM free memory in bytes.

enumerator **ExternalFree**

External PSRAM free memory in bytes.

enumerator **TotalFreePercent**

Total free memory as a percentage.

enumerator **InternalFreePercent**

Internal SRAM free memory as a percentage.

enumerator **ExternalFreePercent**

External PSRAM free memory as a percentage.

enumerator **TotalLargestFreeBlock**

Largest free block across all heaps, in bytes.

enumerator **InternalLargestFreeBlock**

Largest internal SRAM free block, in bytes.

enumerator **ExternalLargestFreeBlock**

Largest external PSRAM free block, in bytes.

enumerator **MinTotalFree**

Minimum total free memory in bytes.

enumerator **MinInternalFree**

Minimum internal SRAM free memory in bytes.

enumerator **MinExternalFree**

Minimum external PSRAM free memory in bytes.

enumerator **MinTotalFreePercent**

Minimum total free memory as a percentage.

enumerator **MinInternalFreePercent**

Minimum internal SRAM free memory as a percentage.

enumerator **MinExternalFreePercent**

Minimum external PSRAM free memory as a percentage.

enumerator **MinTotalLargestFreeBlock**

Minimum largest free block across all heaps, in bytes.

enumerator **MinInternalLargestFreeBlock**

Minimum largest internal SRAM free block, in bytes.

enumerator **MinExternalLargestFreeBlock**

Minimum largest external PSRAM free block, in bytes.

using **ProfilingSignal** = boost::signals2::signal<void(const *ProfileSnapshot*&)>

Signal type emitted for each profiling snapshot.

using **ProfilingSignalSlot** = *ProfilingSignal*::slot_type

Slot type accepted by `connect_profiling_signal()`.

using **ThresholdSignal** = boost::signals2::signal<void(const *ProfileSnapshot*&)>

Signal type emitted when a threshold condition is met.

using **ThresholdSignalSlot** = *ThresholdSignal*::slot_type

Slot type accepted by `connect_threshold_signal()`.

using **SignalConnection** = boost::signals2::scoped_connection

Signal connection type.

Note: This is an RAII smart handle for managing the lifetime of callbacks. When this object is destroyed, the corresponding callback is automatically disconnected. It is recommended to use `std::move()` to transfer ownership for manual management of the connection lifetime.

Public Functions

MemoryProfiler() = default

Construct an idle memory profiler.

MemoryProfiler(const *MemoryProfiler*&) = delete

Copy construction is not supported.

MemoryProfiler &**operator=**(const *MemoryProfiler*&) = delete

Copy assignment is not supported.

MemoryProfiler(*MemoryProfiler*&&) = delete

Move construction is not supported.

MemoryProfiler &**operator=**(*MemoryProfiler*&&) = delete

Move assignment is not supported.

bool **configure_profiling**(const *ProfilingConfig* &config)

Update the profiling configuration.

Parameters **config** –New profiler configuration.
Returns `true` after the configuration is stored.

inline *ProfilingConfig* **get_profiling_config**() const

Get the current profiling configuration.

Returns Active *`ProfilingConfig`*.

bool **start_profiling**(std::shared_ptr<*TaskScheduler*> scheduler, uint32_t period_ms = 0)

Start periodic profiling with a task scheduler.

Parameters
- **scheduler** –Scheduler used to execute periodic sampling.
- **period_ms** –Sampling period in milliseconds. Pass `0` to use `config.sample_interval_ms`.

Returns `true` on success, or `false` if startup fails.

void **stop_profiling**()

Stop periodic profiling.

void **reset_profiling**()

Reset captured snapshots and registered callbacks without changing configuration.

inline bool **is_profiling**() const

Check whether periodic profiling is active.

Returns `true` when a profiling task is active, or `false` otherwise.

inline std::shared_ptr<*ProfileSnapshot*> **get_profiling_latest_snapshot**() const

Get the latest captured snapshot.

Returns Shared pointer to the latest snapshot, or `nullptr` if no snapshot is available.

SignalConnection **connect_profiling_signal**(*ProfilingSignalSlot* slot)

Subscribe to every profiling snapshot.

Parameters **slot** –Callback invoked whenever a new snapshot is produced.
Returns RAII connection handle for the registered callback.

SignalConnection **connect_threshold_signal**(*ThresholdType* type, uint32_t threshold_value, *ThresholdSignalSlot* slot)

Subscribe to threshold events for a specific metric.

Parameters
- **type** –Metric to monitor.
- **threshold_value** –Threshold value interpreted according to `type`.
- **slot** –Callback invoked when the threshold is met by a snapshot.

Returns RAII connection handle for the registered callback.

Public Static Functions

static inline *MemoryProfiler* &**get_instance**()

Get the process-wide singleton instance.

Returns Reference to the singleton profiler.

static std::shared_ptr<*ProfileSnapshot*> **take_snapshot** (*ProfileSnapshot* *last_snapshot = nullptr)

Capture the current memory state.

Parameters **last_snapshot** –Previous snapshot used to update running statistics, or `nullptr` for the first sample.

Returns Shared pointer to the new snapshot, or `nullptr` if allocation fails.

static void **print_snapshot** (const *ProfileSnapshot* &snapshot)

Print a formatted snapshot summary to the log.

Parameters **snapshot** –Snapshot to print.

struct **HeapInfo**

Snapshot of a single heap region.

Public Members

size_t **total_size** = 0

Total heap size in bytes.

size_t **free_size** = 0

Free heap size in bytes.

size_t **largest_free_block** = 0

Size of the largest free block in bytes.

size_t **free_percent** = 0

Free memory ratio expressed as a percentage.

size_t **used_percent** = 0

Used memory ratio expressed as a percentage.

struct **MemoryInfo**

Aggregated memory information for all monitored heap regions.

Public Members

HeapInfo **internal**

Internal SRAM heap information (`MALLOC_CAP_INTERNAL`).

HeapInfo **external**

External PSRAM heap information (`MALLOC_CAP_SPIRAM`).

size_t **total_size** = 0

Total heap size in bytes across all monitored heaps.

size_t **total_free** = 0

Total free memory in bytes across all monitored heaps.

size_t **total_free_percent** = 0

Total free memory ratio expressed as a percentage.

size_t **total_largest_free_block** = 0

Largest free block in bytes across all monitored heaps.

struct **ProfileSnapshot**

Memory profile snapshot captured at one point in time.

Public Members

std::chrono::system_clock::time_point **timestamp**

Snapshot capture time.

MemoryInfo **memory**

Instantaneous memory information.

Statistics **stats**

Running statistics after this snapshot.

struct **ProfilingConfig**

Configuration for periodic memory profiling.

Public Members

uint32_t **sample_interval_ms** = 5000

Sampling interval in milliseconds.

bool **enable_auto_logging** = true

Whether to log each generated snapshot automatically.

struct **Statistics**

Running minimum statistics accumulated across snapshots.

Public Members

size_t **sample_count** = 0

Number of snapshots processed.

size_t **min_total_free** = 0

Minimum total free memory observed, in bytes.

size_t **min_internal_free** = 0

Minimum internal free memory observed, in bytes.

size_t **min_external_free** = 0

Minimum external free memory observed, in bytes.

size_t **min_total_free_percent** = 0

Minimum total free percentage observed.

size_t **min_internal_free_percent** = 0

Minimum internal free percentage observed.

size_t **min_external_free_percent** = 0

Minimum external free percentage observed.

size_t **min_total_largest_free_block** = 0

Minimum total largest free block observed, in bytes.

size_t **min_internal_largest_free_block** = 0

Minimum internal largest free block observed, in bytes.

size_t **min_external_largest_free_block** = 0

Minimum external largest free block observed, in bytes.

Thread Profiler

Public header: `#include "brookesia/lib_utils/thread_profiler.hpp"`

Overview *thread_profiler* samples task and thread state and CPU usage, helping analyze system load and hot tasks.

Features

- Task status, CPU usage, stack info
- Periodic profiling and manual snapshots
- Sorting, filtering, and thresholds
- Optional *TaskScheduler* integration

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/thread_profiler.hpp

Classes

class **ThreadProfiler**

FreeRTOS task profiler for CPU and stack usage monitoring.

This class provides a C++ interface to monitor FreeRTOS task CPU usage, stack usage, and other runtime statistics. It integrates naturally with *TaskScheduler* for periodic profiling.

Public Types

enum class **TaskState**

Task state mirrored from `FreeRTOS eTaskState`.

Values:

enumerator **Running**

`eRunning`.

enumerator **Ready**

`eReady`.

enumerator **Blocked**

`eBlocked`.

enumerator **Suspended**

`eSuspended`.

enumerator **Deleted**

`eDeleted`.

enumerator **Invalid**

`eInvalid`.

enum class **TaskStatus**

Presence status of a task across two samples.

Values:

enumerator **Normal**

Task exists in both samples.

enumerator **Created**

Task appeared during the measurement window.

enumerator **Deleted**

Task disappeared during the measurement window.

enum class **PrimarySortBy**

Optional primary sort criterion.

Values:

enumerator **None**

Disable primary sorting.

enumerator **CoreId**

Sort by CPU core ID first.

enum class **SecondarySortBy**

Secondary sort criterion applied within the primary ordering.

Values:

enumerator **CpuPercent**

Sort by CPU usage percentage in descending order.

enumerator **Priority**

Sort by task priority in descending order.

enumerator **StackUsage**

Sort by stack high-water mark in ascending order.

enumerator **Name**

Sort by task name alphabetically.

enum class **ThresholdType**

Metric used for threshold filtering and callbacks.

Values:

enumerator **CpuPercent**

Filter by CPU usage percentage.

enumerator **Priority**

Filter by task priority.

enumerator **StackUsage**

Filter by stack high-water mark.

using **ProfilingSignalSlot** = std::function<void(const *ProfileSnapshot*&)>

Callback type accepted by `connect_profiling_signal()`.

using **ThresholdSignalSlot** = std::function<void(const std::vector<*TaskInfo*>&)>

Callback type accepted by `connect_threshold_signal()`.

using **ProfilingSignal** = boost::signals2::signal<void(const *ProfileSnapshot*&)>

Signal type emitted for each profiling snapshot.

using **ThresholdSignal** = boost::signals2::signal<void(const std::vector<*TaskInfo*>&)>

Signal type emitted when threshold matches are found.

using **SignalConnection** = boost::signals2::scoped_connection

Signal connection type.

Note: This is an RAII smart handle for managing the lifetime of callbacks. When this object is destroyed, the corresponding callback is automatically disconnected. It is recommended to use `std::move()` to transfer ownership for manual management of the connection lifetime.

Public Functions

ThreadProfiler(const *ThreadProfiler*&) = delete

Copy construction is not supported.

ThreadProfiler &**operator=**(const *ThreadProfiler*&) = delete

Copy assignment is not supported.

ThreadProfiler(*ThreadProfiler*&&) = delete

Move construction is not supported.

ThreadProfiler &**operator=**(*ThreadProfiler*&&) = delete

Move assignment is not supported.

bool **configure_profiling**(const *ProfilingConfig* &config)

Update the profiling configuration.

Parameters **config** –New profiler configuration.
Returns `true` after the configuration is stored.

ProfilingConfig **get_profiling_config**() const

Get the current profiling configuration.

Returns Active *ProfilingConfig*.

bool **start_profiling**(std::shared_ptr<*TaskScheduler*> scheduler, uint32_t sampling_duration_ms = 0, uint32_t profiling_interval_ms = 0)

Start periodic thread profiling using a task scheduler.

Parameters
- **scheduler** –Scheduler used to run the two-stage sampling jobs. The caller must keep it alive until *stop_profiling()* returns.
- **sampling_duration_ms** –Time between the first and second sample in milliseconds. Pass `0` to use `config.sampling_duration_ms`.
- **profiling_interval_ms** –Period between profiling rounds in milliseconds. Pass `0` to use `config.profiling_interval_ms`.

Returns `true` on success, or `false` if profiling cannot be started.

void **stop_profiling**()

Stop periodic profiling.

void **reset_profiling**()

Reset captured data and registered callbacks without changing configuration.

inline bool **is_profiling**() const

Check whether periodic profiling is active.

Returns `true` when profiling tasks are active, or `false` otherwise.

inline std::shared_ptr<*ProfileSnapshot*> **get_profiling_latest_snapshot**()

Get the latest captured profiling snapshot.

Returns Shared pointer to the latest snapshot, or `nullptr` if none is available.

SignalConnection **connect_profiling_signal**(*ProfilingSignalSlot* slot)

Subscribe to every generated profiling snapshot.

Parameters **slot** –Callback invoked whenever a new snapshot is produced.
Returns RAII connection handle for the registered callback.

SignalConnection **connect_threshold_signal**(*ThresholdType* type, uint32_t threshold_value, *ThresholdSignalSlot* slot)

Subscribe to threshold matches for a specific metric.

Parameters
- **type** –Metric to monitor.
- **threshold_value** –Threshold value interpreted according to `type`.
- **slot** –Callback invoked with the tasks that matched the threshold.

Returns RAII connection handle for the registered callback.

Public Static Functions

static inline *ThreadProfiler* &**get_instance**()

Get the process-wide singleton instance.

Returns Reference to the singleton profiler.

static std::shared_ptr<*SampleResult*> **sample_tasks**()

Capture a single raw scheduler sample.

Returns Shared pointer to the captured sample, or `nullptr` on failure.

static std::shared_ptr<*ProfileSnapshot*> **take_snapshot**(const *SampleResult* &start_result, const *SampleResult* &end_result)

Build a profiling snapshot from two raw samples.

Parameters
- **start_result** –Sample taken at the beginning of the measurement window.
- **end_result** –Sample taken at the end of the measurement window.

Returns Shared pointer to the computed snapshot, or `nullptr` on failure.

static void **sort_tasks**(std::vector<*TaskInfo*> &tasks, *PrimarySortBy* primary_sort = *PrimarySortBy*::*CoreId*, *SecondarySortBy* secondary_sort = *SecondarySortBy*::*CpuPercent*)

Sort task entries in place.

Note: The tasks will be sorted by the primary sort criteria first, then by the secondary sort criteria.

Parameters
- **tasks** –Task vector to sort in place.
- **primary_sort** –Optional primary sort criterion.
- **secondary_sort** –Secondary sort criterion.

static void **print_snapshot**(const *ProfileSnapshot* &snapshot, *PrimarySortBy* primary_sort = *PrimarySortBy*::*CoreId*, *SecondarySortBy* secondary_sort = *SecondarySortBy*::*CpuPercent*)

Print a formatted snapshot table to the log.

Parameters
- **snapshot** –Snapshot to print.
- **primary_sort** –Primary sort criterion shown in the first sort column.
- **secondary_sort** –Secondary sort criterion shown in the second sort column.

static bool **get_task_by_name**(const *ProfileSnapshot* &snapshot, const std::string &name, *TaskInfo* &task)

Find a task by name inside a snapshot.

Parameters
- **snapshot** –Snapshot to search.
- **name** –Task name to search for.
- **task** –Output object that receives the matching task info.

Returns `true` if a matching task is found, or `false` otherwise.

static std::vector<*TaskInfo*> **get_tasks_above_threshold**(const *ProfileSnapshot* &snapshot, *ThresholdType* type, uint32_t threshold_value)

Collect tasks whose metric meets a threshold.

Note: For CpuPercent: returns tasks with CPU >= threshold_value For Priority: returns tasks with priority >= threshold_value For StackUsage: returns tasks with stack HWM <= threshold_value (lower = more used)

Parameters

- **snapshot** –Snapshot to inspect.
- **type** –Threshold metric to apply.
- **threshold_value** –Threshold value interpreted according to `type`.

Returns Vector containing every task that satisfies the threshold.

struct **ProfileSnapshot**

Snapshot of all sampled task information.

Public Members

std::chrono::system_clock::time_point **timestamp** = {}

Snapshot capture time.

std::vector<*TaskInfo*> **tasks**

Task data for every sampled task.

Statistics **stats** = {}

Aggregate snapshot statistics.

uint32_t **total_runtime** = 0

Total runtime counter captured in the ending sample.

struct **ProfilingConfig**

Configuration for periodic thread profiling.

Public Members

uint32_t **sampling_duration_ms** = 1000

Delay between the start and end sample, in milliseconds.

uint32_t **profiling_interval_ms** = 5000

Interval between profiling rounds, in milliseconds.

PrimarySortBy **primary_sort** = *PrimarySortBy*::*CoreId*

Primary sort criterion.

SecondarySortBy **secondary_sort** = *SecondarySortBy*::*CpuPercent*

Secondary sort criterion.

bool **enable_auto_logging** = true

Whether to log each generated snapshot automatically.

struct **SampleResult**

Raw sample captured at a single instant.

Public Members

std::chrono::system_clock::time_point **timestamp**

Sample capture time.

std::vector<TaskStatus_t> **task_status**

Raw FreeRTOS task status array.

uint32_t **runtime**

Raw total runtime counter.

struct **Statistics**

Aggregate statistics for a profiling snapshot.

Public Members

size_t **total_tasks** = 0

Total number of tasks in the snapshot.

size_t **running_tasks** = 0

Number of running tasks.

size_t **blocked_tasks** = 0

Number of blocked tasks.

size_t **suspended_tasks** = 0

Number of suspended tasks.

uint32_t **total_cpu_percent** = 0

Aggregate CPU usage percentage.

uint32_t **sample_duration_ms** = 0

Measurement window duration in milliseconds.

struct **TaskInfo**

Profile data for a single FreeRTOS task.

Public Members

std::string **name**
: Task name.

TaskHandle_t **handle** = nullptr
: FreeRTOS task handle.

TaskState **state** = *TaskState*::*Invalid*
: Current task state.

uint32_t **priority** = 0
: Current task priority.

BaseType_t **core_id** = -1
: Bound CPU core ID, or `-1` if unbound.

uint32_t **stack_high_water_mark** = 0
: Stack high-water mark in bytes.

bool **is_stack_external** = false
: Whether the stack is allocated in external RAM.

uint32_t **runtime_counter** = 0
: Raw runtime counter captured in the ending sample.

uint32_t **elapsed_time** = 0
: Runtime counter delta during the measurement window.

uint32_t **cpu_percent** = 0
: CPU usage percentage over the measurement window.

TaskStatus **status** = *TaskStatus*::*Normal*
: Presence status relative to the two samples.

Time Profiler

Public header: `#include "brookesia/lib_utils/time_profiler.hpp"`

Overview *time_profiler* provides hierarchical timing for scopes and events to locate bottlenecks and hot paths.

Features

- Scope and event timing models
- Hierarchical reports with customizable output
- Per-thread sampling
- Convenience macros for instrumentation

> **Warning:** On **ESP32-P4** with `CONFIG_SPIRAM_XIP_FROM_PSRAM=y`, `time_profiler` may crash the system.

API reference

Header File

- utils/brookesia_lib_utils/include/brookesia/lib_utils/time_profiler.hpp

Classes

class **TimeProfiler**

Hierarchical time profiler for scoped code regions and named events.

This class provides hierarchical time profiling capabilities with support for nested scopes, cross-thread events, and detailed statistics reporting.

Public Functions

void **set_format_options**(const *FormatOptions* &options)

Set formatting options used by `report()`.

Parameters **options** –**[in]** Format options to apply.

void **enter_scope**(const std::string &name)

Enter a named profiling scope.

Should be paired with *leave_scope()*. Prefer using BROOKESIA_TIME_PROFILER_SCOPE macro for automatic scope management.

Parameters **name** –**[in]** Scope name to record.

void **leave_scope**()

Leave the current profiling scope.

Records the elapsed time since the matching `enter_scope()` call.

void **start_event**(const std::string &name)

Start timing a named event.

Used for timing events that may span across function boundaries or threads. Must be paired with *end_event()* with the same name.

Parameters **name** –**[in]** Event name to start.

void **end_event**(const std::string &name)

End timing a named event.

Records the elapsed time since the corresponding *start_event()* call.

Parameters **name** –**[in]** Event name to stop.

Statistics **get_statistics**() const

Build a structured snapshot of the collected profiling data.

Returns a structured representation of all profiling data.

Returns `Statistics` structure containing all profiling information.

void **report**()

Generate and log a formatted profiling report.

Prints a hierarchical report of all recorded timings to the log. This method internally calls *get_statistics()* and formats the output.

void **clear**()

Clear all recorded profiling data.

Resets all timing statistics and clears the profiling tree.

Public Static Functions

static inline *TimeProfiler* &**get_instance**()

Get the singleton instance of *TimeProfiler*.

Returns Reference to the singleton *TimeProfiler* instance

struct **FormatOptions**

Formatting options used by *report()*.

Public Types

enum class **SortBy**

Sort order for sibling profiling nodes.

Values:

enumerator **TotalDesc**

Sort by total time descending.

enumerator **NameAsc**

Sort by name ascending.

enumerator **None**

No sorting.

enum class **TimeUnit**

Time unit used for formatted output and exported statistics.

Values:

enumerator **Microseconds**

Display in microseconds.

enumerator **Milliseconds**

Display in milliseconds.

enumerator **Seconds**

Display in seconds.

Public Members

int **name_width** = 32

Width of name column.

int **calls_width** = 6

Width of calls column.

int **num_width** = 10

Width of numeric columns.

int **percent_width** = 7

Width of percentage column.

int **precision** = 2

Decimal precision for numbers.

bool **use_unicode** = false

Use ASCII characters to ensure alignment.

bool **show_percentages** = true

Show percentage columns.

bool **use_color** = false

Use ANSI color codes in output.

SortBy **sort_by** = *SortBy*::*TotalDesc*

Sort order for output.

TimeUnit **time_unit** = *TimeUnit*::*Milliseconds*

Time unit for display.

struct **Node**

Internal tree node representing one profiled scope.

Public Members

std::string **name**

Name of this profiling scope.

double **total** = 0.0

Total time spent in this scope.

size_t **count** = 0

Number of times this scope was entered.

double **min** = std::numeric_limits<double>::infinity()

Minimum time recorded.

double **max** = 0.0

Maximum time recorded.

std::map<std::string, std::unique_ptr<*Node*>> **children**

Child scopes.

Node ***parent** = nullptr

Parent scope.

struct **NodeStatistics**

Aggregated statistics for a single profiling node.

Public Members

std::string **name**

Name of the profiling scope.

size_t **count** = 0

Number of times this scope was entered.

double **total** = 0.0

Total time spent in this scope.

double **self_time** = 0.0

Time spent in this scope excluding children.

double **avg** = 0.0

Average time per call.

double **min** = 0.0

Minimum time recorded.

double **max** = 0.0

Maximum time recorded.

double **pct_parent** = 0.0

Percentage of parent’ s total time.

double **pct_total** = 0.0

Percentage of overall total time.

std::vector<*NodeStatistics*> **children**

Child node statistics.

struct **Statistics**

Structured report returned by *get_statistics()*.

Public Members

std::string **unit_name**

Time unit name (e.g., “ms” , “us” , “s”)

double **overall_total** = 0.0

Overall total time.

std::vector<*NodeStatistics*> **root_children**

Root level node statistics.

class **TimeProfilerScope**

RAII wrapper that profiles the lifetime of a C++ scope.

Automatically calls enter_scope() on construction and leave_scope() on destruction. Use the BROOKESIA_TIME_PROFILER_SCOPE macro for convenient usage.

Public Functions

explicit **TimeProfilerScope**(const std::string &name)

Enter a named profiling scope on construction.

Parameters **name** –**[in]** Scope name to record.

~TimeProfilerScope()

Leave the profiling scope on destruction.

Macros

_BROOKESIA_TIME_PROFILER_CONCAT(a, b)

BROOKESIA_TIME_PROFILER_CONCAT(a, b)

BROOKESIA_TIME_PROFILER_SCOPE(name)

BROOKESIA_TIME_PROFILER_START_EVENT(name)

BROOKESIA_TIME_PROFILER_END_EVENT(name)

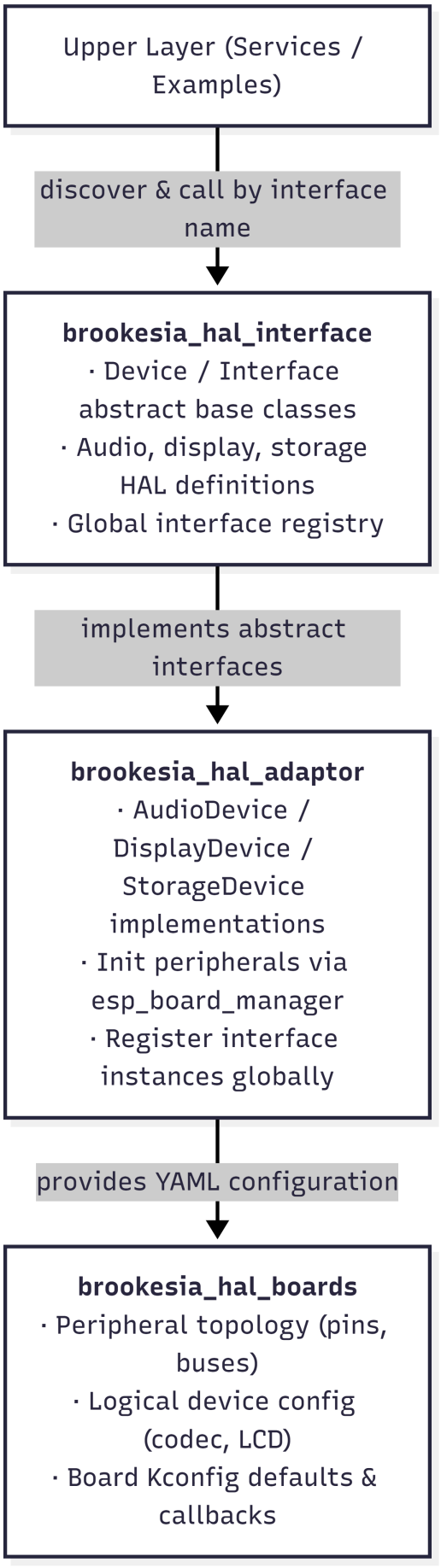
BROOKESIA_TIME_PROFILER_REPORT()

BROOKESIA_TIME_PROFILER_CLEAR()

Chapter 3

HAL Components

ESP-Brookesia HAL consists of three components that work together in layers, bridging the gap between board-level hardware and upper-layer business logic:


Upper Layer (Services / Examples)
discover & call by interface name
brookesia_hal_interface
· Device / Interface abstract base classes
· Audio, display, storage HAL definitions
· Global interface registry
implements abstract interfaces
brookesia_hal_adaptor
· AudioDevice / DisplayDevice / StorageDevice implementations
· Init peripherals via esp_board_manager
· Register interface instances globally
provides YAML configuration
brookesia_hal_boards
· Peripheral topology (pins, buses)
· Logical device config (codec, LCD)
· Board Kconfig defaults & callbacks

- `brookesia_hal_interface`: **Defines abstract interfaces**; upper-layer code depends only on this layer and remains decoupled from hardware details.
- `brookesia_hal_adaptor`: **Implements the abstract interfaces** by reading board configuration via `esp_board_manager` and initializing real peripherals.
- `brookesia_hal_boards`: **Provides board-level YAML configuration** describing the peripheral topology, pin assignments, and driver parameters for each supported board.

Note: Custom boards can be integrated into the ESP-Brookesia HAL framework in two ways:

- **Option 1 (Recommended)**: Create a new board subdirectory under `brookesia_hal_boards/boards/` following the `esp_board_manager` specification and add the required configuration files. No changes to the adaptor layer are needed. See *Add a Custom Board*.
- **Option 2 (Fully Custom)**: Remove the dependencies on `brookesia_hal_adaptor` and `brookesia_hal_boards`, and implement board-level initialization directly against the abstract interfaces in `brookesia_hal_interface`. This is suitable for cases where `esp_board_manager` cannot be used, but requires maintaining compatibility with the interface specification manually.

3.1 HAL Interface

- Component registry: espressif/brookesia_hal_interface
- Public header: `#include "brookesia/hal_interface.hpp"`

3.1.1 Overview

`brookesia_hal_interface` is the hardware abstraction foundation of ESP-Brookesia, providing a unified abstraction between board-level implementations and upper-level subsystems. Its main capabilities include:

- **Device and interface model**: devices aggregate hardware units; interfaces express reusable capabilities (codec playback/recording, display panel/touch/backlight, storage volumes, etc.), with clear separation of concerns
- **Plugin-based registration**: device and interface implementations are registered in a registry and resolved by name at runtime, avoiding hard-coded implementation types in application code
- **Probing and lifecycle**: each device is first probed for availability before initialisation; batch and per-name single-device init/deinit are supported symmetrically
- **Global discovery**: devices can be resolved by plugin name or device logical name; interfaces can be enumerated globally by type or retrieved by name from within a device
- **Built-in HAL declarations**: abstract definitions for common audio, display, and storage interfaces; concrete behaviour is provided by the adaptor layer

3.1.2 Features

Devices and Interfaces

A device represents an independently managed hardware unit (e.g. an audio codec path, a display subsystem, a storage subsystem). Before a device operates it must be **probed**: only when the hardware is available in the current environment does it proceed to initialisation. During initialisation the device collects the **interface instances** it will expose.

An interface represents a stable capability boundary (e.g. codec playback, recording, panel rendering, touch sampling, backlight control, storage medium and filesystem discovery). Multiple instances of the same abstract type can coexist, distinguished by the name used at registration time; that name uniquely identifies one instance in the global interface registry.

Device and interface implementation classes are registered through the project's plugin mechanism; the framework creates or retrieves instances by name at runtime, and upper layers depend only on the abstract types and conventions defined in this component.

Registration and Lifecycle

During batch initialisation, the framework iterates the registry: for each device it executes probe and device-side init in order; on success the framework simultaneously registers that device's declared interfaces into the global interface table. A probe or init failure for one device typically only affects that device; the rest continue to be processed.

Per **plugin registration name** initialisation and deinitialisation are also supported. During deinit the device's interfaces are first removed from the global table, then any device-defined cleanup runs, and finally the device's internal references to interfaces are released.

Typical flow:

1. Registered devices enter the **probe** phase;
2. On success, the device is **initialised** and its interfaces are registered into the global registry; on failure, that **device is skipped**.

Discovery and Naming

Two distinct name types are associated with devices:

Name type	Meaning
Plugin name	Key in the plugin registry, used to look up a device instance directly by registration entry
Device name	The logical name carried by the device object itself; may match or differ from the plugin name

Either path can be used at runtime to resolve the corresponding device.

Interface access is oriented around the **global interface registry**: all currently convertible instances of a given interface type can be listed, or the first matching instance of a type can be retrieved (returned as a name–instance pair). If starting from a device object, a specific interface can also be retrieved from that device's published interface set using the full registration name.

When multiple devices co-exist, it is recommended to add a device-distinguishing prefix or other namespace to interface registration names to avoid global conflicts.

Built-in Capability Scope

The headers provide abstract definitions of common HAL interfaces, covering audio codec playback and recording, display panel and touch and backlight, and storage filesystem discovery. They describe static information, capability parameters, and virtual interface contracts; register operations, bus configuration, and timing are handled by the board-level adaptor or other components.

The following interface headers can be included at once via `brookesia/hal_interface/interfaces.hpp`, or together with the device base class via the aggregation entry `brookesia/hal_interface.hpp`:

Header	Primary types
`audio/codec_player.hpp`	`AudioCodecPlayerIface`
`audio/codec_recorder.hpp`	`AudioCodecRecorderIface`
`display/backlight.hpp`	`DisplayBacklightIface`
`display/panel.hpp`	`DisplayPanelIface`
`display/touch.hpp`	`DisplayTouchIface`
`storage/fs.hpp`	`StorageFsIface`

Interface Classes The component provides the following interface classes:

- `AudioCodecPlayerIface`
- `AudioCodecRecorderIface`
- `DisplayBacklightIface`
- `DisplayPanelIface`
- `DisplayTouchIface`
- `StorageFsIface`

3.1.3 API Reference

Device & Interface Base

Device

Public header: `#include "brookesia/hal_interface/device.hpp"`

API reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/device.hpp

Classes

class **`Device`**

Base class for HAL devices.

A concrete device can publish one or more interface instances through `interfaces_` during initialization, then those interfaces can be queried from the global registry.

Subclassed by *esp_brookesia::hal::AudioDevice*, *esp_brookesia::hal::DisplayDevice*, *esp_brookesia::hal::StorageDevice*

Public Functions

virtual bool **`probe`**`()` = 0

Check whether the device is supported on current runtime conditions.

Returns `true` if the device can be initialized; otherwise `false`.

inline const std::string &**`get_name`**`()` const

Get the registry name of this device.

Returns *Device* name.

`template<typename T> inline requires IsInterface< T > std::shared_ptr< T > get_interface (const std::string &name) const`

Retrieve a typed interface owned by this device by interface registry name.

Template Parameters **`T`** – *Interface* type that derives from *`Interface`*.
Parameters **`name`** – **[in]** Fully-qualified interface name in the registry.
Returns Matching typed interface pointer, or `nullptr` if the name is missing or the type does not match.

Type Aliases

using `esp_brookesia::hal::DeviceRegistry` = lib_utils::*PluginRegistry*<*Device*>

Registry alias for HAL devices.

Interface Base

Public header: `#include "brookesia/hal_interface/interface.hpp"`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/interface.hpp

Classes

class **`Interface`**

Base class for all HAL interfaces exposed by a device.

Each concrete interface derives from this type and provides a stable interface name for runtime lookup via the plugin registry.

Subclassed by *esp_brookesia::hal::AudioCodecPlayerIface*, *esp_brookesia::hal::AudioCodecRecorderIface*, *esp_brookesia::hal::DisplayBacklightIface*, *esp_brookesia::hal::DisplayPanelIface*, *esp_brookesia::hal::DisplayTouchIface*, *esp_brookesia::hal::StorageFsIface*

Public Functions

inline explicit **`Interface`**(std::string_view name)

Construct an interface object with a runtime name.

Parameters **`name`** –**[in]** *Interface* name used by the registry.

virtual **`~Interface`**() = default

Virtual destructor for polymorphic interface usage.

inline std::string_view **`get_name`**() const noexcept

Get the registered interface name.

Returns *Interface* name.

Audio Interface Classes

Codec Player Interface

Public header: `#include "brookesia/hal_interface/audio/codec_player.hpp"`

Class: `AudioCodecPlayerIface`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/audio/codec_player.hpp

Classes

class **AudioCodecPlayerIface** : public esp_brookesia::hal::*Interface*

Playback interface exposed by audio-capable devices.

Public Functions

inline **AudioCodecPlayerIface**(*Info* info)

Construct an audio playback interface.

Parameters **info** –**[in]** Static playback capability information.

virtual **~AudioCodecPlayerIface**() = default

Virtual destructor for polymorphic playback interfaces.

virtual bool **open**(const *Config* &config) = 0

Open the playback backend.

Parameters **config** –**[in]** Dynamic playback configuration.
Returns `true` on success; otherwise `false`.

virtual void **close**() = 0

Close the playback backend.

virtual bool **set_volume**(uint8_t volume) = 0

Set playback volume.

Parameters **volume** –**[in]** Requested output volume percentage.
Returns `true` on success; otherwise `false`.

virtual bool **get_volume**(uint8_t &volume) = 0

Get current playback volume.

Parameters **volume** –**[out]** Current output volume percentage.
Returns `true` on success; otherwise `false`.

virtual bool **mute**() = 0

Set playback to mute.

Note: Calling this interface will mute the playback. Even if the minimum volume is not zero, the volume will be muted to zero.

Returns `true` on success; otherwise `false`.

virtual bool **unmute**() = 0

Unmute playback.

Note: Calling this interface will set the volume to the last set value. If the last set value is zero, the volume will be restored to the default value.

Returns `true` on success; otherwise `false`.

virtual bool **write_data**(const uint8_t *data, size_t size) = 0

Write PCM/encoded payload to the playback backend.

Parameters

- **data** –**[in]** Buffer pointer containing audio payload.
- **size** –**[in]** Buffer size in bytes.

Returns `true` on success; otherwise `false`.

inline const *Info* &**get_info**() const

Get static playback capability information.

Returns Playback information.

Public Static Attributes

static constexpr const char ***NAME** = "AudioCodecPlayer"

Interface registry name.

struct **Config**

Dynamic playback configuration.

Public Members

uint8_t **bits**

Sample bit width.

uint8_t **channels**

Number of output channels.

uint32_t **sample_rate**

Output sample rate in Hz.

struct **Info**

Static playback capability information.

Public Members

uint8_t **volume_default**

Default volume percentage.

uint8_t **volume_min**

Minimum supported volume percentage.

uint8_t **volume_max**

Maximum supported volume percentage.

Codec Recorder Interface

Public header: `#include "brookesia/hal_interface/audio/codec_recorder.hpp"`

Class: `AudioCodecRecorderIface`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/audio/codec_recorder.hpp

Classes

class **AudioCodecRecorderIface** : public esp_brookesia::hal::*Interface*

Recording interface exposed by audio-capable devices.

Public Functions

inline **AudioCodecRecorderIface**(*Info* info)

Construct an audio recording interface.

Parameters **info** –**[in]** Static recording capability information.

virtual **~AudioCodecRecorderIface**() = default

Virtual destructor for polymorphic recording interfaces.

virtual bool **open**() = 0

Open the recording backend.

Returns `true` on success; otherwise `false`.

virtual void **close**() = 0

Close the recording backend.

virtual bool **read_data**(uint8_t *data, size_t size) = 0

Read captured audio payload.

Parameters
- **data** –**[out]** Destination buffer for captured bytes.
- **size** –**[in]** Requested byte count.

Returns `true` on success; otherwise `false`.

virtual bool **set_general_gain**(float gain) = 0

Set the general gain.

Parameters **gain** –**[in]** The gain to set.
Returns `true` on success; otherwise `false`.

virtual bool **set_channel_gains**(const std::map<uint8_t, float> &gains) = 0

Set the channel gains.

Parameters **gains** –**[in]** The channel gains to set.
Returns `true` on success; otherwise `false`.

inline const *Info* &**get_info**() const

Get static recording capability information.

Returns Recording information.

Public Static Attributes

static constexpr const char ***NAME** = "AudioCodecRecorder"

Interface registry name.

struct **Info**

Static recording capability information.

Public Members

uint8_t **bits**

Sample bit width.

uint8_t **channels**

Number of capture channels.

uint32_t **sample_rate**

Capture sample rate in Hz.

std::string **mic_layout**

Microphone layout descriptor.

float **general_gain**

Global input gain.

std::map<uint8_t, float> **channel_gains**

Per-channel gain overrides.

Display Interface Classes

Display Panel Interface

Public header: `#include "brookesia/hal_interface/display/panel.hpp"`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/display/panel.hpp

Classes

class **DisplayPanelIface** : public esp_brookesia::hal::*Interface*

Display panel interface for rendering pixel data.

Public Types

enum class **PixelFormat** : uint8_t

Pixel format enum.

Values:

enumerator **RGB565**

enumerator **RGB888**

enumerator **Max**

enum class **BusType** : uint8_t

Driver-specific bus type enum.

Values:

enumerator **Generic**

enumerator **RGB**

< Generic bus type, such as SPI, I80, QSPI, etc.

enumerator **MIPI**

< RGB bus type.

enumerator **Max**

< MIPI bus type.

Public Functions

inline **DisplayPanelIface**(*Info* info)

Construct a display panel interface.

Parameters **info** –**[in]** Static panel capability information.

virtual **~DisplayPanelIface**() = default

Virtual destructor for polymorphic panel interfaces.

virtual bool **draw_bitmap**(uint32_t x1, uint32_t y1, uint32_t x2, uint32_t y2, const uint8_t *data) = 0

Draw a bitmap into a rectangular panel region.

The region follows half-open bounds semantics: `[x1, x2)` and `[y1, y2)`.

Parameters

- **x1** –**[in]** Left coordinate.
- **y1** –**[in]** Top coordinate.
- **x2** –**[in]** Right coordinate (exclusive).
- **y2** –**[in]** Bottom coordinate (exclusive).
- **data** –**[in]** Pixel buffer pointer.

Returns `true` on success; otherwise `false`.

inline virtual bool **get_driver_specific**(*DriverSpecific* &specific)

Get the driver-specific data.

Parameters **specific** –**[out]** The driver-specific data.
Returns `true` on success; otherwise `false`.

inline const *Info* &**get_info**() const

Get static panel capability information.

Returns Panel information.

Public Static Attributes

static constexpr const char ***NAME** = "DisplayPanel"

Interface registry name.

struct **DriverSpecific**

Driver-specific data.

Public Members

void *`io_handle` = nullptr

Handle of the IO.

void *`panel_handle` = nullptr

Handle of the panel.

BusType `bus_type` = *BusType*::*Max*

Bus type.

struct `Info`

Static panel capability information.

Public Functions

inline uint8_t `get_pixel_bits`() const

Get the number of bits per pixel.

Returns Number of bits per pixel.

inline bool `is_valid`() const

Check if the panel information is valid.

Returns `true` if the panel information is valid; otherwise `false`.

Public Members

uint16_t `h_res` = 0

Horizontal resolution in pixels.

uint16_t `v_res` = 0

Vertical resolution in pixels.

PixelFormat `pixel_format` = *PixelFormat*::*Max*

Pixel format.

Touch Interface

Public header: `#include "brookesia/hal_interface/display/touch.hpp"`

Class: `DisplayTouchIface`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/display/touch.hpp

Classes

class `DisplayTouchIface` : public esp_brookesia::hal::*Interface*

Touch-input interface paired with a display.

Public Types

enum class **OperationMode** : uint8_t

Supported event acquisition mode.

Values:

enumerator **Polling**

Read touch points by polling.

enumerator **Interrupt**

Read touch points by interrupt.

enumerator **Max**

Public Functions

inline **DisplayTouchIface**(*Info* info)

Construct a touch-input interface.

Parameters **info** –**[in]** Static touch capability information.

virtual **~DisplayTouchIface**() = default

Virtual destructor for polymorphic touch interfaces.

virtual bool **read_points**(std::vector<*Point*> &points) = 0

Read current touch points.

Parameters **points** –**[out]** Output touch points.
Returns `true` on success; otherwise `false`.

virtual bool **register_interrupt_handler**(InterruptHandler handler) = 0

Register an interrupt handler.

Parameters **handler** –**[in]** Interrupt handler. If `nullptr`, the interrupt handler will be unregistered.
Returns `true` on success; otherwise `false`.

inline virtual bool **get_driver_specific**(*DriverSpecific* &specific)

Get the driver-specific data.

Parameters **specific** –**[out]** The driver-specific data.
Returns `true` on success; otherwise `false`.

inline const *Info* &**get_info**() const

Get static touch capability information.

Returns Touch information.

Public Static Attributes

static constexpr const char ***NAME** = "DisplayTouch"

Interface registry name.

struct **DriverSpecific**

Driver-specific data.

Public Members

void *`io_handle` = nullptr

Handle of the IO.

void *`touch_handle` = nullptr

Handle of the touch.

struct `Info`

Static touch capability information.

Public Members

uint16_t `x_max` = 0

Maximum X coordinate value.

uint16_t `y_max` = 0

Maximum Y coordinate value.

OperationMode `operation_mode` = *OperationMode*::*Max*

Supported acquisition mode.

struct `Point`

A single touch point sample.

Public Members

int16_t `x`

X coordinate.

int16_t `y`

Y coordinate.

uint16_t `pressure`

Pressure or strength value from controller.

Backlight Interface

Public header: `#include "brookesia/hal_interface/display/backlight.hpp"`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/display/backlight.hpp

Classes

class **DisplayBacklightIface** : public esp_brookesia::hal::*Interface*

Display backlight control interface.

Public Functions

inline **DisplayBacklightIface**(*Info* info)

Construct a display backlight interface.

Parameters **info** –**[in]** Static backlight capability information.

virtual **~DisplayBacklightIface**() = default

Virtual destructor for polymorphic backlight interfaces.

virtual bool **set_brightness**(uint8_t percent) = 0

Set backlight brightness.

Parameters **percent** –**[in]** Brightness percentage.
Returns `true` on success; otherwise `false`.

virtual bool **get_brightness**(uint8_t &percent) = 0

Get current backlight brightness.

Parameters **percent** –**[out]** Current brightness percentage.
Returns `true` on success; otherwise `false`.

virtual bool **turn_on**() = 0

Turn on the backlight.

Note: Calling this interface will restore the brightness to the last set value. If the last set value is zero, the brightness will be restored to the default value.

Returns `true` on success; otherwise `false`.

virtual bool **turn_off**() = 0

Turn off the backlight.

Note: Calling this interface will turn off the backlight. Even if the minimum brightness is not zero, the brightness will be turned off to zero.

Returns `true` on success; otherwise `false`.

inline const *Info* &**get_info**() const

Get static backlight capability information.

Returns Backlight information.

Public Static Attributes

static constexpr const char ***NAME** = "DisplayBacklight"

Interface registry name.

struct **Info**

Static backlight capability information.

Public Members

uint8_t **brightness_default**

Default brightness percentage.

uint8_t **brightness_min**

Minimum brightness percentage.

uint8_t **brightness_max**

Maximum brightness percentage.

Storage Interface Classes

Filesystem Interface

Public header: `#include "brookesia/hal_interface/storage/fs.hpp"`

Class: `StorageFsIface`

API Reference

Header File

- hal/brookesia_hal_interface/include/brookesia/hal_interface/storage/fs.hpp

Classes

class **StorageFsIface** : public esp_brookesia::hal::*Interface*

File-system discovery interface for storage-capable devices.

Public Types

enum class **MediumType**

Supported storage medium type.

Values:

enumerator **Flash**

Flash medium.

enumerator **SDCard**

SD / TF card medium.

enum class **FileSystemType**

Supported file-system type.

Values:

enumerator **SPIFFS**

SPIFFS file system.

enumerator **FATFS**

FAT file system.

enumerator **LittleFS**

LittleFS file system.

Public Functions

inline **StorageFsIface**()

Construct a storage file-system interface.

virtual **~StorageFsIface**() = default

Virtual destructor for polymorphic storage interfaces.

inline const std::vector<*Info*> &**get_all_info**() const

Enumerate all mounted file-system entries.

Returns Collection of file-system metadata entries.

Public Static Attributes

static constexpr const char ***NAME** = "StorageFs"

Interface registry name.

struct **Info**

Metadata for one mounted file-system entry.

Public Members

FileSystemType **fs_type**

File-system type.

MediumType **medium_type**

Storage medium type.

const char ***mount_point**

Mount point.

3.2 HAL Adaptor

- Component registry: espressif/brookesia_hal_adaptor
- Public header: `#include "brookesia/hal_adaptor.hpp"`

3.2.1 Overview

`brookesia_hal_adaptor` is the board-level HAL adaptor of ESP-Brookesia. Based on the device/interface model in *HAL Interface*, it initialises real peripherals via `esp_board_manager` and registers **audio**, **display**, and **storage** capabilities into the global HAL table for upper layers to discover by name.

3.2.2 Features

Built-in Devices

The component ships three board-level devices, each registered as a singleton; after initialisation they publish their interfaces into the global table:

Device class (logical name)	Registered interface implementations	Notes
`AudioDevice` (`"Audio"`)	`AudioCodecPlayerIface` (`CODEC_PLAYER_IMPL_NAME`), `AudioCodecRecorderIface` (`CODEC_RECORDER_IMPL_NAME`)	Playback via board Audio DAC; recording via Audio ADC. Each sub-implementation can be disabled in Kconfig; requires board capability `ESP_BOARD_DEV_AUDIO_CODEC_SUPPORT`.
`DisplayDevice` (`"Display"`)	`DisplayBacklightIface` (`LEDC_BACKLIGHT_IMPL_NAME`), `DisplayPanelIface` (`LCD_PANEL_IMPL_NAME`), `DisplayTouchIface` (`LCD_TOUCH_IMPL_NAME`)	LEDC backlight, LCD panel, and I2C touch can each be disabled; require `ESP_BOARD_DEV_LEDC_CTRL_SUPPORT`, `ESP_BOARD_DEV_DISPLAY_LCD_SUPPORT`, `ESP_BOARD_DEV_LCD_TOUCH_I2C_SUPPORT` respectively.
`StorageDevice` (`"Storage"`)	`StorageFsIface` (`GENERAL_FS_IMPL_NAME`)	General filesystem implementation; supports SPIFFS (`ESP_BOARD_DEV_FS_SPIFFS_SUPPORT`) and SD card / FATFS (`ESP_BOARD_DEV_FS_FAT_SUPPORT`), enabled per Kconfig.

Configuration

Each sub-interface can be enabled or disabled individually under **ESP-Brookesia: Hal Adaptor Configurations** in `menuconfig`; default capability parameters (volume range, recording format, backlight range, etc.) are also adjustable there, and are mapped to compile-time macros by `macro_configs.h`.

To override default capability parameters before initialisation, call `set_codec_player_info`, `set_codec_recorder_info`, or `set_ledc_backlight_info` on the corresponding device singleton. Calls after initialisation typically have no effect.

3.2.3 API Reference

Header File

- hal/brookesia_hal_adaptor/include/brookesia/hal_adaptor/display/device.hpp

Classes

class **`DisplayDevice`** : public esp_brookesia::hal::*Device*

Board-backed display device: registers panel, touch, and backlight HAL interfaces after board bring-up.

Obtained via *get_instance()*. Not copyable or movable.

Public Functions

bool **set_ledc_backlight_info**(*DisplayBacklightIface*::*Info* info)

Overrides default static backlight capability information used when constructing the LEDC backlight implementation.

Parameters **info** –**[in]** Backlight capability descriptor (brightness range and default).
Returns `true` if the value was stored; `false` on invalid input or if backlight is already initialized.

Public Static Functions

static inline *DisplayDevice* &**get_instance**()

Returns the process-wide singleton display device.

Returns Reference to the unique *DisplayDevice* instance.

Public Static Attributes

static constexpr const char ***DEVICE_NAME** = "Display"

Logical device name passed to the base *Device* constructor.

static constexpr const char ***LEDC_BACKLIGHT_IMPL_NAME** = "Display:LedcBacklight"

Registry key for the LEDC-based backlight HAL implementation (`"Display:LedcBacklight"`).

static constexpr const char ***LCD_PANEL_IMPL_NAME** = "Display:LcdPanel"

Registry key for the LCD panel HAL implementation (`"Display:LcdPanel"`).

static constexpr const char ***LCD_TOUCH_IMPL_NAME** = "Display:LcdTouch"

Registry key for the LCD touch HAL implementation (`"Display:LcdTouch"`).

Header File

- hal/brookesia_hal_adaptor/include/brookesia/hal_adaptor/audio/device.hpp

Classes

class **AudioDevice** : public esp_brookesia::hal::*Device*

Board-backed audio device: registers codec player and recorder HAL interfaces after board bring-up.

Obtained via *get_instance()*. Not copyable or movable.

Public Functions

bool **set_codec_player_info**(*AudioCodecPlayerIface*::*Info* info)

Overrides default static playback capability information used when constructing the codec player implementation.

Parameters **info** –**[in]** Codec player capability descriptor (volume range and default).
Returns `true` if the value was stored; `false` on invalid input or if the player is already initialized.

bool **set_codec_recorder_info**(*AudioCodecRecorderIface*::*Info* info)

Overrides default static recording capability information used when constructing the codec recorder implementation.

Parameters `info` –**[in]** Codec recorder capability descriptor (format, channels, gains, etc.).
Returns `true` if the value was stored; `false` on invalid input or if the recorder is already initialized.

Public Static Functions

static inline *AudioDevice* &**get_instance**()

Returns the process-wide singleton audio device.

Returns Reference to the unique *AudioDevice* instance.

Public Static Attributes

static constexpr const char ***DEVICE_NAME** = "Audio"

Logical device name passed to the base *Device* constructor.

static constexpr const char ***CODEC_PLAYER_IMPL_NAME** = "Audio:CodecPlayer"

Registry key for the codec player HAL implementation (`"Audio:CodecPlayer"`).

static constexpr const char ***CODEC_RECORDER_IMPL_NAME** = "Audio:CodecRecorder"

Registry key for the codec recorder HAL implementation (`"Audio:CodecRecorder"`).

Header File

- hal/brookesia_hal_adaptor/include/brookesia/hal_adaptor/storage/device.hpp

Classes

class **StorageDevice** : public esp_brookesia::hal::*Device*

Board-backed storage device: publishes a general filesystem HAL interface after bring-up.

Obtained via *get_instance()*. Not copyable or movable.

Public Static Functions

static inline *StorageDevice* &**get_instance**()

Returns the process-wide singleton storage device.

Returns Reference to the unique *StorageDevice* instance.

Public Static Attributes

static constexpr const char ***DEVICE_NAME** = "Storage"

Logical device name passed to the base *Device* constructor.

static constexpr const char ***GENERAL_FS_IMPL_NAME** = "Storage:GenralFS"

Registry key for the general filesystem HAL interface (`"Storage:GenralFS"`).

3.3 HAL Boards

- Component Registry: espressif/brookesia_hal_boards

3.3.1 Overview

`brookesia_hal_boards` is the board configuration collection for ESP-Brookesia. It uses YAML files to describe the peripheral topology and device parameters for each supported board, allowing *HAL Adaptor* to initialize hardware at runtime without any board-specific hardcoding.

3.3.2 Supported Boards

Board Name	Chip	Description
`esp_vocat_board_v1_0`	ESP32-S3	ESP VoCat Board V1.0 —AI Pet Companion Development Board
`esp_vocat_board_v1_2`	ESP32-S3	ESP VoCat Board V1.2 —AI Pet Companion Development Board
`esp_box_3`	ESP32-S3	ESP-BOX-3 Development Board
`esp32_s3_korvo2_v3`	ESP32-S3	ESP32-S3-Korvo-2 V3 Development Board
`esp32_p4_function_ev`	ESP32-P4	ESP32-P4-Function-EV-Board
`esp_sensair_shuttle`	ESP32-C5	ESP Sensair Shuttle Module

3.3.3 Directory Structure

Each board has its own subdirectory under `boards/<board-name>/` containing the following files:

```
boards/
└── <board>/
    ├── board_info.yaml          # Board metadata (name, chip, version,␣
↪manufacturer, etc.)
    ├── board_devices.yaml       # Logical device configurations (audio codec, LCD,
↪ touch, storage, etc.)
    ├── board_peripherals.yaml   # Low-level peripheral configurations (I2C/I2S/
↪SPI buses, GPIO, LEDC, etc.)
    ├── sdkconfig.defaults.board # Board-specific Kconfig defaults (Flash, PSRAM,␣
↪etc.)
    └── setup_device.c           # Board-specific device factory callbacks (for␣
↪custom driver initialization)
```

Device Types

`board_devices.yaml` describes the logical functional modules on the board. Common device types include:

Device Type	Description
`audio_codec`	Audio codec chip (DAC playback / ADC recording); supports ES8311, ES7210, internal ADC, etc.
`display_lcd`	LCD display; supports SPI (ST77916, ILI9341), DSI (EK79007), and other interfaces
`lcd_touch`	Touch panel; supports CST816S, GT911, and other I2C touch controllers
`ledc_ctrl`	PWM backlight control via LEDC
`fs_fat` / `fs_spiffs`	File system storage; supports SD card (SDMMC/SPI) and SPIFFS
`camera`	Camera (CSI interface)
`power_ctrl`	GPIO-based power control (audio power, LCD/SD card power, etc.)
`gpio_ctrl`	General-purpose GPIO control (LEDs, buttons, etc.)

Peripheral Configuration

`board_peripherals.yaml` describes the pin assignments and parameters for low-level hardware resources:

- **I2C**: SDA/SCL pins, port number
- **I2S**: MCLK/BCLK/WS/DOUT/DIN pins, sample rate, bit depth
- **SPI**: MOSI/MISO/CLK/CS pins, SPI host, transfer size
- **LEDC**: Backlight GPIO, PWM frequency and resolution
- **GPIO**: Power control, amplifier enable, LED, and other standalone pin configurations

`sdkconfig.defaults.board` contains board-specific Kconfig defaults tightly coupled to the hardware, such as Flash size, PSRAM mode and speed, CPU frequency, and `brookesia_hal_adaptor` audio recording parameters.

If a driver requires a custom initialization flow (e.g., injecting vendor-specific register sequences into an LCD driver), this is handled through factory callbacks in `setup_device.c`.

3.3.4 Usage

Select a Board

Run the following command in the project root directory, specifying the target board name:

```
idf.py gen-bmgr-config -b <board>
```

Available `<board>` values are listed in *Supported Boards*. If `brookesia_hal_boards` is included as a local path dependency, specify the `boards/` directory with the `-c` option:

```
idf.py gen-bmgr-config -b <board> -c path/to/brookesia_hal_boards/boards
```

Note: In example projects that use `idf_ext.py`, the `-c` argument is injected automatically at build time and does not need to be added manually.

Add a Custom Board

Create a new board subdirectory under `boards/` (or any custom directory) and add the following files in order:

1. **board_info.yaml**: Fill in the board name, chip, version, and description.
2. **board_peripherals.yaml**: Configure peripherals according to the actual pin and bus assignments.
3. **board_devices.yaml**: Describe the on-board devices with their types and configurations.
4. **sdkconfig.defaults.board**: Add board-specific Kconfig defaults.
5. **setup_device.c** *(optional)*: Implement factory functions if the driver requires extra initialization steps.

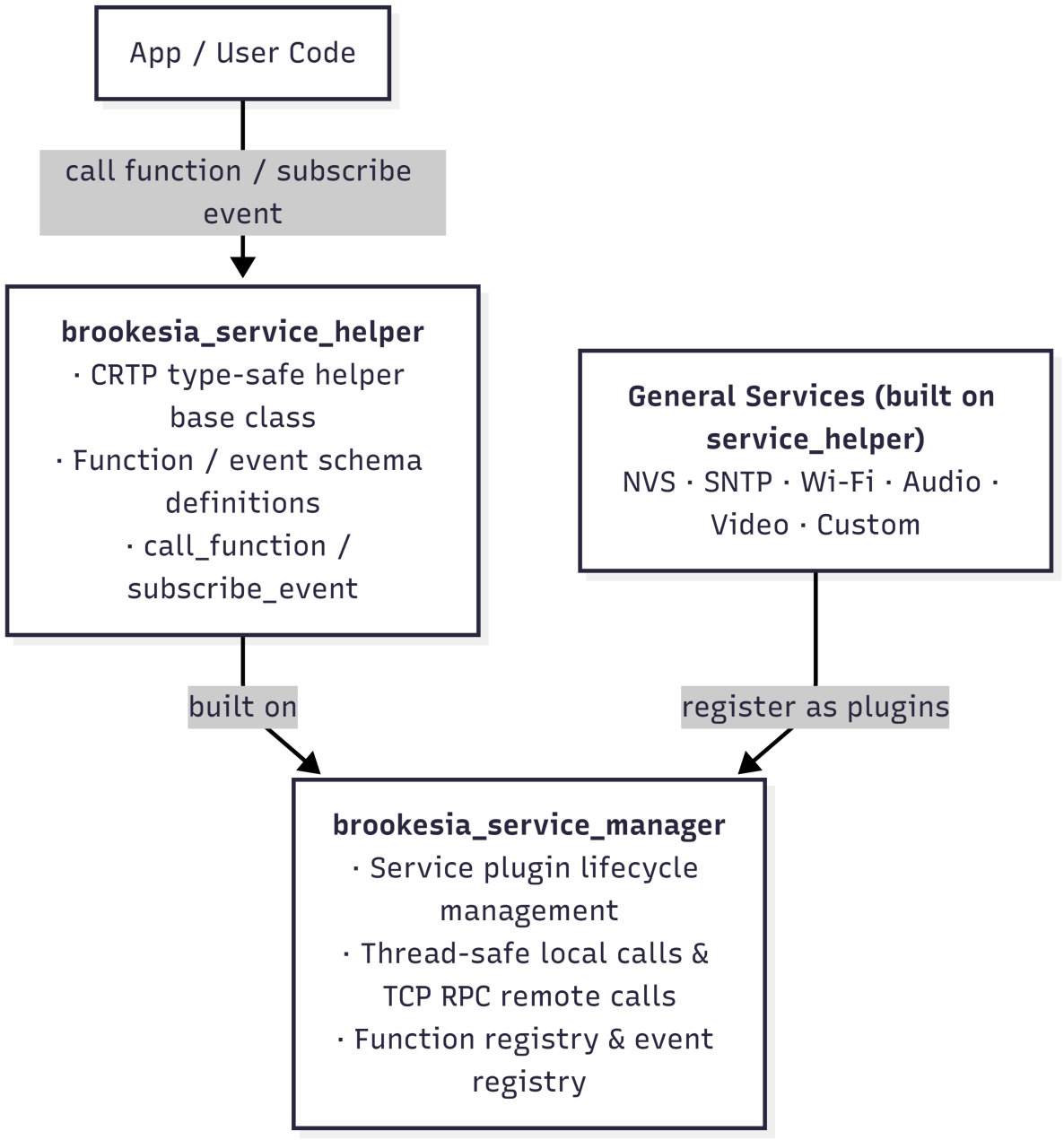
Once done, run `idf.py gen-bmgr-config -b <new_board>` to use the new board.

Note: For the complete `esp_board_manager` configuration format reference, see the esp_board_manager component documentation.

Chapter 4

Service Components

This section documents the ESP-Brookesia service framework components. The framework consists of a service framework layer and a general-services layer. Their component hierarchy is shown below:



- `brookesia_service_manager`: The service framework core, responsible for plugin registration, function routing, and event dispatching in both local and RPC communication modes.
- `brookesia_service_helper`: A CRTP-based type-safe helper layer that simplifies service function/event definition and invocation.
- **General Services**: Concrete services implemented on top of `service_helper`, registered into `service_manager` and discoverable by name from upper layers.

4.1 Service framework

4.1.1 Service Manager

- Component registry: espressif/brookesia_service_manager
- Public header: `#include "brookesia/service_manager.hpp"`

Overview

brookesia_service_manager is the core service framework: lifecycle management, function and event registration, local calls, and remote RPC.

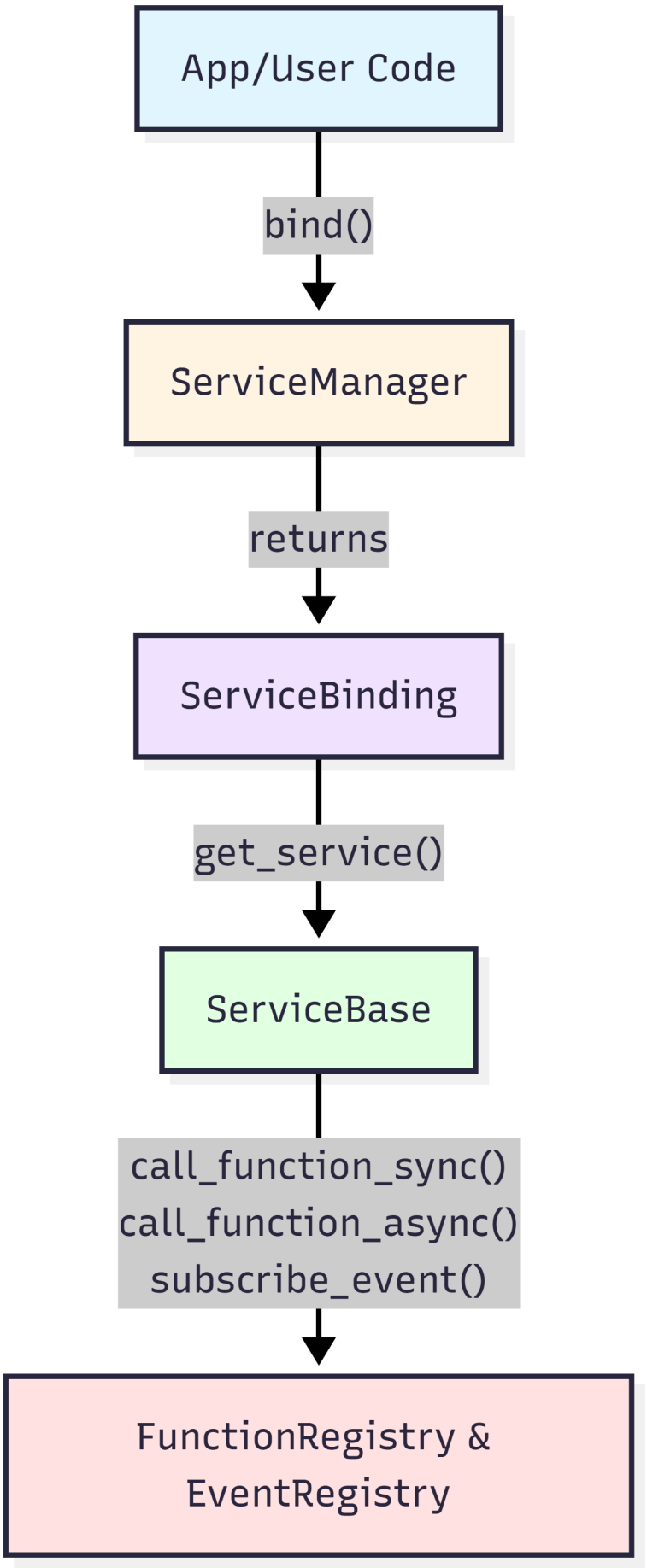
Features

- **Lifecycle**: Centralized init, start, stop, and deinit.
- **Dual mode**: High-performance local calls and TCP/JSON RPC.
- **Unified model**: Function definitions, registries, events, and subscriptions.
- **Thread safety**: Local runners and schedulers for safe execution.
- **Decoupling**: Apps call through a stable API without depending on provider details.

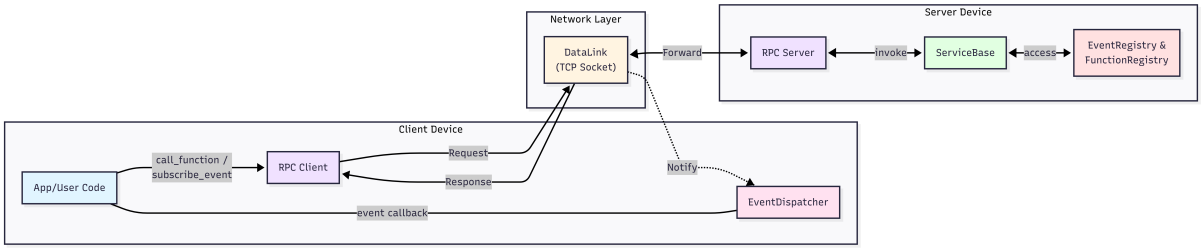
Communication architecture

brookesia_service_manager supports **local** and **remote RPC** modes.

Local mode After *ServiceManager* binds a service, the app uses *ServiceBase* to access function and event registries with minimal overhead.


App/User Code
bind()
ServiceManager
returns
ServiceBinding
get_service()
ServiceBase
call_function_sync()
call_function_async()
subscribe_event()
FunctionRegistry &
EventRegistry

Remote RPC mode Clients use TCP sockets and JSON to reach remote services—suitable for cross-device or cross-language use.



Local vs remote RPC

Item	Local (*ServiceBase*)	Remote RPC (*rpc::Client*)
Deployment	Same device	Cross-device
Transport	Direct calls	TCP + JSON
Latency	Low (ms)	Higher (network)
Performance	No serialization	Serialize/deserialize
Call rate	High frequency	Medium/low frequency
Thread safety	Async scheduling & guards	Network isolation
Languages	C++	Language-agnostic
Network	Not required	LAN or routable network
Typical use	In-device collaboration	Cross-device / cross-language calls

Modules

Service runtime Lifecycle, dispatch, binding, and RPC integration—built on `ServiceBase`, `ServiceManager`, and `LocalTestRunner`.

API reference

Function system Defines callable interfaces, validates parameters, and dispatches to handlers—function model and registry.

API reference

Event system Event definitions, validation, and dispatch to local subscribers or RPC subscribers—definitions, registry, and dispatcher.

API reference

RPC TCP + JSON exposure of functions and events: protocol, data link, bridging, client, and server.

API reference

Common Shared types and macros for the service manager.

API reference

Service runtime

API reference

Service base Public header: `#include "brookesia/service_manager/service/base.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/service/base.hpp

Classes

class **ServiceBase**

Base class for bindable services managed by *ServiceManager*.

Subclasses expose callable functions and publishable events through the function and event registries owned by the service.

Subclassed by *esp_brookesia::agent::Base*, *esp_brookesia::agent::Manager*, esp_brookesia::expression::Emote, esp_brookesia::service::Audio, *esp_brookesia::service::CustomService*, esp_brookesia::service::NVS, esp_brookesia::service::SNTP, esp_brookesia::service::VideoDecoder, esp_brookesia::service::VideoEncoder, esp_brookesia::service::wifi::Wifi

Public Types

using **FunctionHandlerMap** = std::map<std::string, FunctionHandler>

Map from function names to service-side handlers.

using **FunctionResultHandler** = std::function<void(FunctionResult&&)>

Callback invoked with the result of an asynchronous function call.

Public Functions

inline **ServiceBase**(const *Attributes* &attributes)

Construct a service with immutable attributes.

Parameters **attributes** –**[in]** Public metadata and scheduler preferences for the service.

virtual **~ServiceBase**()

Virtual destructor.

inline virtual std::vector<FunctionSchema> **get_function_schemas**()

Get the function schemas list.

Subclasses should override this method to return an array of function schemas

```
std::vector<FunctionSchema> get_function_schemas() override {
    return {
        {
            "add", "Add numbers", {
                {"a", "First", FunctionValueType::Number},
                {"b", "Second", FunctionValueType::Number}
            }
        },
        {
            "sub", "Subtract", {
                {"a", "First", FunctionValueType::Number},
                {"b", "Second", FunctionValueType::Number}
            }
```

(continues on next page)

(continued from previous page)

```
        }
    };
}
```

Returns std::vector<FunctionSchema> List of function schemas

inline virtual std::vector<EventSchema> **get_event_schemas**()

Get the event schemas list.

Subclasses should override this method to return an array of event schemas

```
std::vector<EventSchema> get_event_schemas() override {
    return {
        {
            "value_change", "Value changed", {
                {"value", "New value", EventItemType::Number}
            }
        }
    };
}
```

Returns std::vector<EventSchema> List of event schemas

bool **call_function_async**(const std::string &name, FunctionParameterMap parameters_map, *FunctionResultHandler* handler = nullptr)

Call a function asynchronously with parameters map (non-blocking)

Parameters

- **name** –**[in]** Function name to call
- **parameters_map** –**[in]** FunctionParameterMap map (key-value pairs)
- **handler** –**[in]** FunctionResultHandler to handle the result, if not provided, the result will be ignored

Returns true if called successfully, false otherwise

bool **call_function_async**(const std::string &name, std::vector<FunctionValue> parameters_values, *FunctionResultHandler* handler = nullptr)

Call a function asynchronously with parameters values (non-blocking)

Parameters

- **name** –**[in]** Function name to call
- **parameters_values** –**[in]** FunctionParameterMap values (ordered array)
- **handler** –**[in]** FunctionResultHandler to handle the result, if not provided, the result will be ignored

Returns true if called successfully, false otherwise

bool **call_function_async**(const std::string &name, const boost::json::object &parameters_json, *FunctionResultHandler* handler = nullptr)

Call a function asynchronously with JSON parameters (non-blocking)

Parameters

- **name** –**[in]** Function name to call
- **parameters_json** –**[in]** FunctionParameterMap in JSON object format
- **handler** –**[in]** FunctionResultHandler to handle the result, if not provided, the result will be ignored

Returns true if called successfully, false otherwise

FunctionResult **call_function_sync**(const std::string &name, FunctionParameterMap parameters_map, uint32_t timeout_ms = BROOKESIA_SERVICE_MANAGER_DEFAULT_CALL_FUNCTION_TIMEOUT_MS)

Call a function synchronously with parameters map (blocking with timeout)

Parameters
- **name** –**[in]** Function name to call
- **parameters_map** –**[in]** FunctionParameterMap map (key-value pairs)
- **timeout_ms** –**[in]** Timeout in milliseconds (default: 100ms)

Returns FunctionResult Result of the function call

FunctionResult **call_function_sync**(const std::string &name, std::vector<FunctionValue> parameters_values, uint32_t timeout_ms = BROOKESIA_SERVICE_MANAGER_DEFAULT_CALL_FUNCTION_TIMEOUT_MS)

Call a function synchronously with parameters values (blocking with timeout)

Parameters
- **name** –**[in]** Function name to call
- **parameters_values** –**[in]** FunctionParameterMap values (ordered array)
- **timeout_ms** –**[in]** Timeout in milliseconds (default: 100ms)

Returns FunctionResult Result of the function call

FunctionResult **call_function_sync**(const std::string &name, const boost::json::object &parameters_json, uint32_t timeout_ms = BROOKESIA_SERVICE_MANAGER_DEFAULT_CALL_FUNCTION_TIMEOUT_MS)

Call a function synchronously with JSON parameters (blocking with timeout)

Parameters
- **name** –**[in]** Function name to call
- **parameters_json** –**[in]** FunctionParameterMap in JSON object format
- **timeout_ms** –**[in]** Timeout in milliseconds (default: 100ms)

Returns FunctionResult Result of the function call

EventRegistry::*SignalConnection* **subscribe_event**(const std::string &event_name, const *EventRegistry*::*SignalSlot* &slot)

Subscribe to an event.

Parameters
- **event_name** –**[in]** Event name to subscribe
- **slot** –**[in]** Callback slot to be invoked when event is published

Returns *EventRegistry::SignalConnection* RAII scoped connection object for managing the subscription, automatically disconnects the subscription when the connection object is destroyed.

inline bool **is_initialized**() const

Check if the service is initialized.

Returns true if initialized, false otherwise

inline bool **is_running**() const

Check if the service is running.

Returns true if running, false otherwise

inline bool **is_server_connected**() const

Check if the service is connected to server.

Returns true if connected, false otherwise

inline const *Attributes* &**get_attributes**() const

Get the service attributes.

Returns const *Attributes*& Reference to service attributes

inline virtual std::string **get_call_task_group**() const

Get the call task group name.

Returns std::string Call task group name

inline virtual std::string **get_event_task_group**() const

Get the event task group name.

Returns std::string Event task group name

inline virtual std::string **get_request_task_group**() const

Get the request task group name.

Returns std::string Request task group name

Public Static Functions

template<typename **T**>
static inline FunctionResult **to_function_result**(std::expected<*T*, std::string> result)

Helper function to convert std::expected to FunctionResult.

Template Parameters **T** –Return value type, can be void or any type convertible to FunctionValue
Parameters **result** –**[in]** std::expected object
Returns FunctionResult Converted result

struct **Attributes**

Service attributes configuration.

Public Functions

inline bool **has_scheduler**() const

Check whether a dedicated task scheduler configuration is present.
Returns true if the service should create its own scheduler.

inline const lib_utils::*TaskScheduler*::*StartConfig* &**get_scheduler_config**() const

Get the dedicated task scheduler configuration.

Note: Call this only when *has_scheduler()* returns `true`.

Returns const *lib_utils::TaskScheduler::StartConfig*& Config stored in `task_scheduler_config`.

Public Members

std::string **name**

Service name.

std::vector<std::string> **dependencies** = {}

Optional: List of dependent service names, will be started in order.

std::optional<lib_utils::*TaskScheduler*::*StartConfig*> **task_scheduler_config** = std::nullopt

Optional: Task scheduler configuration. If configured, service request tasks will be scheduled to this scheduler; otherwise, *ServiceManager*' s scheduler will be used

bool **bindable** = true

Optional: Whether the service can be bound.

Macros

BROOKESIA_SERVICE_FUNC_HANDLER_0(func_name, func_call)

BROOKESIA_SERVICE_FUNC_HANDLER_1(func_name, param_name, param_type, func_call)

BROOKESIA_SERVICE_FUNC_HANDLER_2(func_name, param1_name, param1_type, param2_name, param2_type, func_call)

BROOKESIA_SERVICE_FUNC_HANDLER_3(func_name, p1_name, p1_type, p2_name, p2_type, p3_name, p3_type, func_call)

Service manager

Public header: `#include "brookesia/service_manager/service/manager.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/service/manager.hpp

Classes

class **ServiceBinding**

Service binding handle.

RAII wrapper for service binding that automatically releases the service and its dependencies when the binding goes out of scope.

Public Functions

inline bool **is_valid**() const

Check if the binding is valid.

Returns true if valid and service is running, false otherwise

inline explicit **operator bool**() const

Explicit conversion to bool.

Returns true if valid, false otherwise

inline std::shared_ptr<*ServiceBase*> **get_service**() const

Get the service object.

Returns std::shared_ptr<ServiceBase> Pointer to the service

inline std::shared_ptr<*ServiceBase*> **get_dependency_service**(const std::string &name) const

Get a dependency service by name.

Parameters **name** –**[in]** Dependency service name

Returns std::shared_ptr<ServiceBase> Pointer to the dependency service, or nullptr if not found

void **release**()

Release the service binding.

class **ServiceManager**

Service manager singleton.

Manages service lifecycle, dependencies, and RPC communication.

Public Functions

bool **init**()

Initialize the service manager.

Returns true if initialized successfully, false otherwise

void **deinit**()

Deinitialize the service manager.

bool **start**(const lib_utils::*TaskScheduler*::*StartConfig* &config = DEFAULT_TASK_SCHEDULER_START_CONFIG)

Start the service manager.

Parameters **config** –**[in]** Task scheduler start configuration
Returns true if started successfully, false otherwise

void **stop**()

Stop the service manager.

bool **add_service**(std::shared_ptr<*ServiceBase*> service)

Add a service to the service manager.

Parameters **service** –**[in]** Service to add
Returns true if added successfully, false otherwise

bool **remove_service**(const std::string &name)

Remove a service by name.

Parameters **name** –**[in]** Service name
Returns true if removed successfully, false otherwise

ServiceBinding **bind**(const std::string &name)

Bind a service by name.

Parameters **name** –**[in]** Service name
Returns *ServiceBinding* Service binding handle

bool **start_rpc_server**(const rpc::*Server*::*Config* &config = rpc::*Server*::*Config*(), uint32_t timeout_ms = 100)

Start the RPC server.

Parameters
- **config** –**[in]** RPC server configuration
- **timeout_ms** –**[in]** Timeout in milliseconds (default: 100ms)

Returns true if started successfully, false otherwise

void **stop_rpc_server**()

Stop the RPC server.

bool **connect_rpc_server_to_services**(std::vector<std::string> names = {})

Connect RPC server to services.

Parameters **names** –**[in]** Service names to connect (empty means all services)
Returns true if connected successfully, false otherwise

bool **disconnect_rpc_server_from_services**(std::vector<std::string> names = {})

Disconnect RPC server from services.

Parameters **names** –**[in]** Service names to disconnect (empty means all services)
Returns true if disconnected successfully, false otherwise

std::shared_ptr<rpc::*Client*> **new_rpc_client** (const *RPC_ClientConfig* &config = *RPC_ClientConfig*())

Create a new RPC client.

Parameters **config** –**[in]** RPC client configuration
Returns std::shared_ptr<rpc::Client> Shared pointer to the RPC client

FunctionResult **call_rpc_function_sync** (std::string host, const std::string &service_name, const std::string &function_name, boost::json::object params, uint32_t timeout_ms = BROOKESIA_SERVICE_MANAGER_RPC_CLIENT_CALL_FUNCTION_TIMEO uint16_t port = BROOKESIA_SERVICE_MANAGER_RPC_SERVER_LISTEN_PORT)

Call an RPC function synchronously.

Parameters
- **host** –**[in]** Host address
- **service_name** –**[in]** Service name
- **function_name** –**[in]** Function name
- **params** –**[in]** Function parameters in JSON format
- **timeout_ms** –**[in]** Timeout in milliseconds (default: configured timeout)
- **port** –**[in]** Server port (default: configured port)

Returns FunctionResult Result of the function call

inline bool **is_initialized** () const

Check if the service manager is initialized.

Returns true if initialized, false otherwise

inline bool **is_running** () const

Check if the service manager is running.

Returns true if running, false otherwise

inline bool **is_rpc_server_running** () const

Check if the RPC server is running.

Returns true if running, false otherwise

inline std::shared_ptr<*ServiceBase*> **get_service** (const std::string &name)

Get a service by name.

Parameters **name** –**[in]** Service name
Returns std::shared_ptr<ServiceBase> Pointer to the service, or nullptr if not found

Public Static Functions

static inline lib_utils::*TaskScheduler*::*StartConfig* **make_default_task_scheduler_start_config** ()

Default worker configuration used by `start()`.

The configuration creates two worker threads for service dispatching and uses the module-level scheduling defaults defined in `macro_configs.h`.

static inline *ServiceManager* &**get_instance** ()

Get the singleton instance.

Returns *ServiceManager*& Reference to the singleton instance

struct **RPC_ClientConfig**

Optional callbacks applied to RPC clients created by the manager.

Public Members

rpc::*Client*::*DisconnectCallback* **on_disconnect_callback**

Called after the RPC transport disconnects.

rpc::*Client*::*DeinitCallback* **on_deinit_callback**

Called when the client is deinitialized.

Local test runner Public header: `#include "brookesia/service_manager/service/local_runner.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/service/local_runner.hpp

Classes

class **LocalTestRunner**

Local service test runner.

A test framework based on TaskScheduler that supports executing test sequences in order, where each test item can specify a start delay and run duration.

Public Functions

bool **run_tests**(const *RunTestsConfig* &config, const std::vector<LocalTestItem> &test_items)

Run test sequence.

Parameters
- **config** –**[in]** Run tests configuration
- **test_items** –**[in]** List of test items

Returns true if all tests passed, false otherwise

inline bool **run_tests**(const std::string &service_name, const std::vector<LocalTestItem> &test_items)

Run test sequence with default configuration.

Parameters
- **service_name** –**[in]** Service name to test
- **test_items** –**[in]** List of test items

Returns true if all tests passed, false otherwise

const std::vector<bool> &**get_results**() const

Get test results.

Returns const std::vector<bool>& Result of each test item

struct **RunTestsConfig**

Function system

API reference

Function definition Public header: `#include "brookesia/service_manager/function/definition.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/function/definition.hpp

Function registry Public header: `#include "brookesia/service_manager/function/registry.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/function/registry.hpp

Classes

class **FunctionRegistry**

Registry of callable service functions and their schemas.

Public Functions

bool **add**(FunctionSchema func_schema, FunctionHandler func_handler)

Register a function schema and its handler.

Parameters
- **func_schema** –**[in]** Function metadata.
- **func_handler** –**[in]** Handler that executes the function.

Returns true on success, false if the function name already exists.

bool **remove**(const std::string &func_name)

Remove a registered function.

Parameters **func_name** –**[in]** Function name to remove.
Returns true if the function was removed.

bool **remove_all**()

Remove every registered function.

Returns true if the registry was cleared successfully.

FunctionResult **call**(const std::string &func_name, FunctionParameterMap parameters)

Validate parameters and invoke a registered function.

Parameters
- **func_name** –**[in]** Function name to call.
- **parameters** –**[in]** Parameter map passed to the handler.

Returns FunctionResult Execution result or validation failure information.

inline const FunctionSchema ***get_schema**(const std::string &func_name)

Look up the schema for a registered function.

Parameters **func_name** –**[in]** Function name to query.
Returns const FunctionSchema* Pointer to the schema, or `nullptr` if not found.

std::vector<FunctionSchema> **get_schemas**() const

Get a snapshot of all registered function schemas.

Returns std::vector<FunctionSchema> Copy of the registered schemas.

boost::json::array **get_schemas_json**()

Export all registered function schemas as JSON.

Returns boost::json::array JSON representation of every schema.

inline bool **has**(const std::string &func_name)

Check whether a function is registered.

Parameters **func_name** –**[in]** Function name to check.
Returns true if the function exists.

inline size_t **get_count**()

Get the number of registered functions.

Returns size_t Count of registered functions.

Event system

API reference Public header: `#include "brookesia/service_manager/event/definition.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/event/definition.hpp

Event dispatcher Public header: `#include "brookesia/service_manager/event/dispatcher.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/event/dispatcher.hpp

Classes

class **EventDispatcher**

Dispatches RPC event notifications to local subscription callbacks.

Public Types

using **NotifyCallback** = std::function<void(const EventItemMap&)>

Callback invoked when a subscribed event notification arrives.

Public Functions

bool **subscribe**(const std::string &subscription_id, *NotifyCallback* callback)

Register a callback for a subscription identifier.

Parameters
- **subscription_id** –**[in]** Subscription id returned by the RPC server.
- **callback** –**[in]** Callback to invoke when matching notifications arrive.

Returns true on success, false if the subscription id is already registered.

void **unsubscribe**(const std::string &subscription_id)

Remove a previously registered subscription callback.

Parameters **subscription_id** –**[in]** Subscription id to remove.

void **on_notify**(const std::vector<std::string> &subscription_ids, const EventItemMap &event_items)

Dispatch an incoming notification to all matching local callbacks.

Parameters

- **subscription_ids** –**[in]** Subscription ids targeted by the notification.
- **event_items** –**[in]** Event payload associated with the notification.

Event registry Public header: `#include "brookesia/service_manager/event/registry.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/event/registry.hpp

Classes

class **EventRegistry**

Registry of event schemas and active RPC subscriptions for one service.

Public Types

using **Subscriptions** = std::unordered_set<std::string>

Set of subscription identifiers attached to an event.

using **Signal** = boost::signals2::signal<void(const std::string &event_name, const EventItemMap &event_items)>

Signal type used for local in-process event listeners.

using **SignalConnection** = boost::signals2::scoped_connection

RAII connection handle returned by `Signal::connect()`.

using **SignalSlot** = *Signal*::slot_type

Slot type accepted by `subscribe_event()`.

Public Functions

bool **add**(EventSchema event_schema)

Register an event schema.

Parameters **event_schema** –**[in]** Schema to add.
Returns true on success, false if an event with the same name already exists.

void **remove**(const std::string &event_name)

Remove a registered event schema.

Parameters **event_name** –**[in]** Name of the event to remove.

void **remove_all**()

Remove all registered event schemas.

bool **validate_items**(const std::string &event_name, const EventItemMap &event_items)

Validate an event payload against a registered schema.

Parameters

- **event_name** –**[in]** Event name to validate against.

- **event_items** –**[in]** Event payload to validate.

Returns true if the payload conforms to the schema, false otherwise.

bool **on_rpc_subscribe**(const std::string &event_name, std::string &subscription_id, std::string &error_message)

Create a new RPC subscription for the given event.

Parameters
- **event_name** –**[in]** Name of the event to subscribe to.
- **subscription_id** –**[out]** Generated subscription id on success.
- **error_message** –**[out]** Validation or registration failure details.

Returns true if the subscription was created, false otherwise.

void **on_rpc_unsubscribe_by_name**(const std::string &event_name)

Remove all RPC subscriptions associated with an event name.

Parameters **event_name** –**[in]** Name of the event whose subscriptions should be removed.

void **on_rpc_unsubscribe_by_subscriptions**(const *Subscriptions* &subscriptions)

Remove a set of RPC subscriptions from every registered event.

Parameters **subscriptions** –**[in]** Subscription ids to remove.

inline const EventSchema ***get_schema**(const std::string &event_name)

Look up the schema for a registered event.

Parameters **event_name** –**[in]** Event name to query.
Returns const EventSchema* Pointer to the schema, or `nullptr` if not found.

std::vector<EventSchema> **get_schemas**() const

Get a snapshot of all registered event schemas.

Returns std::vector<EventSchema> Copy of the registered schemas.

boost::json::array **get_schemas_json**()

Export all registered event schemas as JSON.

Returns boost::json::array JSON representation of every schema.

inline bool **has**(const std::string &event_name)

Check whether an event schema exists.

Parameters **event_name** –**[in]** Event name to check.
Returns true if the event is registered.

inline size_t **get_count**()

Get the number of registered event schemas.

Returns size_t Count of registered events.

Subscriptions **get_subscriptions**(const std::string &event_name)

Get all RPC subscription ids associated with an event.

Parameters **event_name** –**[in]** Event name to query.
Returns Subscriptions Copy of the registered subscription ids.

Signal ***get_signal**(const std::string &event_name)

Get the in-process signal associated with an event.

Parameters **event_name** –**[in]** Event name to query.
Returns Signal* Pointer to the signal, or `nullptr` if the event does not exist.

RPC

API reference

RPC protocol Public header: `#include "brookesia/service_manager/rpc/protocol.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/protocol.hpp

RPC connection Public header: `#include "brookesia/service_manager/rpc/connection.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/connection.hpp

Classes

class **ServerConnection**

Bridges one service' s registries to the RPC server transport.

Public Types

using **Responder** = std::function<bool(size_t, Response&&)>

Callback used to send an RPC response back to a client connection.

using **Notifier** = std::function<bool(std::size_t, Notify&&)>

Callback used to push an RPC notification to a client connection.

using **RequestHandler** = std::function<bool(size_t, std::string&&, std::string&&, FunctionParameterMap&&)>

Optional callback that overrides request handling for custom routing.

Public Functions

inline **ServerConnection**(std::string name, *FunctionRegistry* &function_registry, *EventRegistry* &event_registry)

Construct a server-side connection wrapper for one service.

Parameters

- **name** –**[in]** Service name exposed to RPC clients.
- **function_registry** –**[in]** Registry used for function calls.
- **event_registry** –**[in]** Registry used for event subscriptions and notifications.

inline void **set_responder**(*Responder* responder)

Set the responder used for sending RPC responses.

Parameters **responder** –**[in]** Transport callback.

inline void **set_notifier**(*Notifier* notifier)

Set the notifier used for sending RPC notifications.

Parameters **notifier** –**[in]** Transport callback.

inline void **set_request_handler**(*RequestHandler* request_handler)

Set a custom request handler.

Parameters **request_handler** –**[in]** Custom transport-aware request handler.

inline void **activate**(bool active)

Enable or disable request handling on this connection.

Parameters **active** –**[in]** `true` to accept RPC traffic, `false` to reject it.

std::expected<std::shared_ptr<FunctionResult>, std::string> **on_request**(std::string &&request_id, size_t connection_id, std::string &&method, FunctionParameterMap &&parameters)

Process an incoming RPC request for this service.

Parameters

- **request_id** –**[in]** RPC request id.
- **connection_id** –**[in]** Transport connection id.
- **method** –**[in]** Requested method name.
- **parameters** –**[in]** Method parameters.

Returns std::expected<std::shared_ptr<FunctionResult>, std::string> Function result on success, or an error.

void **on_connection_closed**(size_t connection_id)

Clear per-connection subscription state when a client disconnects.

Parameters **connection_id** –**[in]** Closed transport connection id.

bool **publish_event**(const std::string &event_name, const EventItemMap &event_items)

Publish a service event to subscribed RPC clients.

Parameters

- **event_name** –**[in]** Event name to publish.
- **event_items** –**[in]** Event payload.

Returns true if notifications were dispatched successfully.

bool **respond_request**(size_t connection_id, Response &&response)

Send an RPC response to one client connection.

Parameters

- **connection_id** –**[in]** Transport connection id.
- **response** –**[in]** Response to send.

Returns true if the response was handed to the transport.

RPC data link base Public header: `#include "brookesia/service_manager/rpc/data_link_base.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/data_link_base.hpp

Classes

class **DataLinkBase**

Abstract TCP data-link layer shared by RPC clients and servers.

Subclassed by *esp_brookesia::service::rpc::DataLinkClient*, *esp_brookesia::service::rpc::DataLinkServer*

Public Types

using **OnDataReceived** = std::function<void(const std::string &data, size_t connection_id)>

Callback invoked when a complete payload string is received.

using **OnConnectionEstablished** = std::function<void(size_t connection_id)>

Callback invoked when a transport connection becomes active.

using **OnConnectionClosed** = std::function<void(size_t connection_id)>

Callback invoked when a transport connection closes.

Public Functions

inline **DataLinkBase**(boost::asio::io_context::executor_type executor)

Construct the base transport with an executor.

Parameters **executor** –**[in]** Executor that owns asynchronous I/O operations.

inline void **set_on_data_received**(*OnDataReceived* callback)

Set the callback used for incoming payloads.

Parameters **callback** –**[in]** Callback to install.

inline void **set_on_connection_established**(*OnConnectionEstablished* callback)

Set the callback used for successful connection establishment.

Parameters **callback** –**[in]** Callback to install.

inline void **set_on_connection_closed**(*OnConnectionClosed* callback)

Set the callback used for connection teardown.

Parameters **callback** –**[in]** Callback to install.

Public Static Functions

static inline size_t **get_active_global_sockets_count**()

Get the number of active sockets across all data-link instances.

Returns size_t Current global socket count.

static inline size_t **get_max_global_sockets_count**()

Get the configured global socket cap.

Returns size_t Maximum number of sockets allowed globally.

static inline bool **is_global_sockets_limit_reached**()

Check whether the global socket cap has been reached.

Returns true if no more sockets should be opened.

static inline size_t **get_max_active_global_sockets_count**()

Get the maximum number of simultaneously active sockets observed.

Returns size_t Historical peak socket count.

RPC data link client

Public header: `#include "brookesia/service_manager/rpc/data_link_client.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/data_link_client.hpp

Classes

class **DataLinkClient** : public esp_brookesia::service::rpc::*DataLinkBase*

TCP transport used by RPC clients.

Public Functions

inline **DataLinkClient**(boost::asio::io_context::executor_type executor)

Construct a client transport on the given executor.

Parameters **executor** –**[in]** Executor used for socket operations.

~DataLinkClient() override

Destructor.

bool **connect**(const std::string &host, uint16_t port, size_t timeout_ms)

Connect to a remote server.

Parameters

- **host** –**[in]** Remote host name or address.
- **port** –**[in]** Remote TCP port.
- **timeout_ms** –**[in]** Connection timeout in milliseconds.

Returns true if the connection succeeds.

bool **disconnect**()

Disconnect the active client connection.

Returns true if a connection existed and cleanup succeeded.

bool **send_data**(std::string &&data)

Send one payload over the active connection.

Parameters **data** –**[in]** Framed payload to send.
Returns true if the send operation was queued successfully.

bool **is_connected**()

Check whether the client currently has an active connection.

Returns true if connected.

RPC data link server Public header: `#include "brookesia/service_manager/rpc/data_link_server.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/data_link_server.hpp

Classes

class **DataLinkServer** : public esp_brookesia::service::rpc::*DataLinkBase*

TCP transport used by the RPC server to accept multiple clients.

Public Functions

inline explicit **DataLinkServer**(boost::asio::io_context::executor_type executor, size_t max_connections)

Construct a server transport.

Parameters

- **executor** –**[in]** Executor used for socket operations.
- **max_connections** –**[in]** Maximum number of concurrent client connections.

~DataLinkServer() override

Destructor.

bool **start**(uint16_t port, size_t timeout_ms)

Start listening for client connections.

Parameters
- **port** –**[in]** Local TCP port.
- **timeout_ms** –**[in]** Startup timeout in milliseconds.

Returns true if the server starts successfully.

void **stop**()

Stop accepting clients and close all active connections.

bool **send_data**(size_t connection_id, std::string &&data)

Send one payload to a connected client.

Parameters
- **connection_id** –**[in]** *Client* connection id.
- **data** –**[in]** Framed payload to send.

Returns true if the payload was queued successfully.

size_t **get_active_connections_count**()

Get the number of currently active client connections.

Returns size_t Active connection count.

std::vector<size_t> **get_active_connection_ids**()

Get the ids of all active client connections.

Returns std::vector<size_t> Snapshot of active connection ids.

inline bool **is_running**()

Check whether the server transport is currently accepting traffic.

Returns true if running.

inline size_t **get_max_connections_count**()

Get the configured maximum number of client connections.

Returns size_t Connection cap.

inline bool **is_connection_limit_reached**()

Check whether the local connection cap has been reached.

Returns true if no more clients should be accepted.

inline size_t **get_max_active_connections_count**()

Get the peak number of simultaneously active client connections.

Returns size_t Historical maximum for this server instance.

RPC client Public header: `#include "brookesia/service_manager/rpc/client.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/client.hpp

Classes

class **Client**

RPC client used to call service functions and subscribe to service events.

Public Types

using **DeinitCallback** = std::function<void()>

Callback invoked when the client is deinitialized.

using **DisconnectCallback** = std::function<void()>

Callback invoked after the transport disconnects.

Public Functions

inline **Client**(*DeinitCallback* on_deinit_callback = nullptr)

Construct a client with an optional deinitialization callback.

Parameters **on_deinit_callback** –**[in]** Callback invoked by *deinit()*.

~Client()

Destructor.

bool **init**(boost::asio::io_context::executor_type executor, *DisconnectCallback* on_disconnect_callback)

Initialize the client transport on an executor.

Parameters

- **executor** –**[in]** Executor used for async socket operations.
- **on_disconnect_callback** –**[in]** Callback invoked if the transport disconnects.

Returns true if initialization succeeds.

void **deinit**()

Deinitialize the client and release its transport resources.

bool **connect**(const std::string &host, uint16_t port, uint32_t timeout_ms)

Connect to an RPC server.

Parameters

- **host** –**[in]** Remote host name or address.
- **port** –**[in]** Remote port.
- **timeout_ms** –**[in]** Connection timeout in milliseconds.

Returns true if the connection succeeds.

void **disconnect**()

Disconnect from the current RPC server.

std::future<FunctionResult> **call_function_async**(const std::string &target, const std::string &method, boost::json::object &&params)

Call a remote service function asynchronously.

Parameters

- **target** –**[in]** Target service name.
- **method** –**[in]** Remote function name.
- **params** –**[in]** JSON object containing function parameters.

Returns std::future<FunctionResult> Future resolved with the call result.

FunctionResult **call_function_sync**(const std::string &target, const std::string &method, boost::json::object &&params, size_t timeout_ms)

Call a remote service function synchronously.

Parameters
- **target** –**[in]** Target service name.
- **method** –**[in]** Remote function name.
- **params** –**[in]** JSON object containing function parameters.
- **timeout_ms** –**[in]** Wait timeout in milliseconds.

Returns FunctionResult Call result or timeout/error information.

std::string **subscribe_event**(const std::string &target, const std::string &event_name, *EventDispatcher*::*NotifyCallback* callback, size_t timeout_ms)

Subscribe to a remote service event.

Parameters
- **target** –**[in]** Target service name.
- **event_name** –**[in]** Event name to subscribe to.
- **callback** –**[in]** Callback invoked when notifications arrive.
- **timeout_ms** –**[in]** RPC timeout in milliseconds.

Returns std::string Subscription id, or an empty string on failure.

bool **unsubscribe_events**(const std::string &target, const std::vector<std::string> &subscription_ids, size_t timeout_ms)

Unsubscribe from multiple remote event subscriptions.

Parameters
- **target** –**[in]** Target service name.
- **subscription_ids** –**[in]** Subscription ids to cancel.
- **timeout_ms** –**[in]** RPC timeout in milliseconds.

Returns true if the unsubscribe request succeeds.

RPC server

Public header: `#include "brookesia/service_manager/rpc/server.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/server.hpp

Classes

class **Server**

RPC server that exposes registered services over TCP.

Public Functions

inline **Server**(boost::asio::io_context::executor_type executor, const *Config* &config = *Config*())

Construct an RPC server on the given executor.

Parameters
- **executor** –**[in]** Executor used for socket operations.
- **config** –**[in]** Transport configuration.

~Server()

Destructor.

bool **init**()

Initialize internal transport resources.

Returns true if initialization succeeds.

void **deinit**()

Deinitialize the server and release transport resources.

bool **start**(uint32_t timeout_ms)

Start listening for RPC clients.

Parameters **timeout_ms** –**[in]** Startup timeout in milliseconds.
Returns true if the server starts successfully.

void **stop**()

Stop the RPC server.

bool **add_connection**(std::shared_ptr<*ServerConnection*> connection)

Attach a service connection to the server.

Parameters **connection** –**[in]** Service connection wrapper to expose.
Returns true if the connection is added successfully.

bool **remove_connection**(const std::string &name)

Remove a service connection by service name.

Parameters **name** –**[in]** Service name to remove.
Returns true if a matching connection was removed.

std::shared_ptr<*ServerConnection*> **get_connection**(const std::string &name)

Get a registered service connection.

Parameters **name** –**[in]** Service name to query.
Returns std::shared_ptr<ServerConnection> Matching connection, or `nullptr`.

struct **Config**

Runtime configuration for the RPC server transport.

Public Members

uint16_t **listen_port**

TCP port used for listening.

size_t **max_connections**

Maximum number of simultaneous client connections.

Common

API reference Public header: `#include "brookesia/service_manager/common.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/common.hpp

4.1.2 Service Helper

- Component registry: espressif/brookesia_service_helper
- Public header: `#include "brookesia/service_helper.hpp"`

Overview

brookesia_service_helper is the unified application-facing helper layer for Brookesia services: typed wrappers and schema entry points.

Features

- **Single entry**: Helpers encapsulate each service to simplify integration.
- **Schema-friendly**: Works with function/event schemas for standardized calls and docs.
- **Decoupling**: Apps target helpers instead of concrete providers.
- **Extensible**: New services can add helpers and documentation over time.

Modules

Base

- Public header: `#include "brookesia/service_helper/base.hpp"`

Overview *esp_brookesia::service::helper::Base<Derived>* is the CRTP base for Service Helpers, providing unified function calls, event subscriptions, and schema validation.

Features

- **CRTP contract**: *DerivedMeta* requires *FunctionId* / *EventId*, *get_name*, *get_*_schemas*, etc.
- **Calls**: *call_function_sync* / *call_function_async* with packing and parsing.
- **Events**: Type-safe *subscribe_event* overloads.
- **Availability**: *is_available*, *get_function_schema*, *get_event_schema*.
- **Timeouts**: Optional *Timeout(ms)* on sync calls.
- **Macros**: *BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_** for handler boilerplate.

Usage Derive a helper with function/event enums and schemas:

```
class MyHelper : public esp_brookesia::service::helper::Base<MyHelper> {
public:
    enum class FunctionId { Ping };
    enum class EventId { Ready };
    static std::string_view get_name();
    static std::vector<FunctionSchema> get_function_schemas();
    static std::vector<EventSchema> get_event_schemas();
};
```

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/base.hpp

Classes

template<typename **Derived**>

class **Base**

Base class for all service helpers (CRTP)

Note: Concept check is removed from template parameter to allow CRTP pattern where Derived is incomplete. Type and method checks are performed when methods are actually used via static_assert or compiler errors.

Template Parameters **Derived** – The derived class (must be a subclass of *Base*)

Public Functions

template<typename EventIdType, typename Callable> inline requires (has_event_name_first_param_v< Callable > &&! has_event_items_second_param_v< Callable >) static EventRegistry

Subscribe to an event with automatic parameter extraction.

```
// Event with two schema parameters: (string name, double value)
auto conn = subscribe_event(EventId::ValueChanged,
    [](const std::string &event_name, const std::string &name, double␣
↪value) {
        std::cout << event_name << ": " << name << " = " << value <<␣
↪std::endl;
    });

// Event with no schema parameters, only event_name
auto conn = subscribe_event(EventId::Ready,
    [](const std::string &event_name) {
        std::cout << event_name << " triggered!" << std::endl;
    });

// Using string_view for event_name
auto conn = subscribe_event(EventId::StatusChanged,
    [](std::string_view event_name, bool status) {
        std::cout << event_name << ": " << (status ? "ON" : "OFF") <<␣
↪std::endl;
    });
```

Note: REQUIRED: First parameter must be `std::string` or `std::string_view` to receive `event_name`.

Note: Second parameter must not be `EventItemMap`; use the other overload for that case.

Note: Remaining parameters are extracted from `EventItemMap` based on the event-schema order.

Note: All non-`event_name` parameter types must satisfy the `ConvertibleToEventItem` concept.

Note: Parameter types must match the actual `EventItem` types in the map.

Template Parameters

- **EventIdType** –Type of the event identifier (enum)
- **Callable** –Type of the callable object (lambda, function, std::function, etc.) Event identifier is passed as `event_id`. Callable object is passed as `callback`, with first parameter as event_name and remaining parameters matching the event schema order.

Returns *EventRegistry::SignalConnection* Connection object (scoped, automatically unsubscribes on destruction)

Public Static Functions

template<typename **ReturnType**>
static inline std::expected<*ReturnType*, std::string> **process_function_result**(const FunctionResult &result)

Helper function to process function result and convert to expected return type.

Parameters **result** –**[in]** Function result from service call
Returns std::expected containing ReturnType or error message

template<typename **ReturnType** = void, typename **FunctionIdType**, typename ...**Args**>
static inline std::expected<*ReturnType*, std::string> **call_function_sync**(*FunctionIdType* function_id, *Args*&&... args)

Call a function synchronously with variadic arguments (timeout at end)

```
// No arguments with default timeout
auto result = call_function_sync<int>(FunctionId::GetValue);

// With arguments and default timeout
auto result = call_function_sync<int>(FunctionId::Add, 1.0, 2.0);

// With custom timeout at the end
auto result = call_function_sync<int>(FunctionId::Add, 1.0, 2.0,␣
↪Timeout(500));
```

Note: Arguments are packed into std::vector<FunctionValue> in the order provided

Note: Last argument can be Timeout(milliseconds) to specify custom timeout

Note: All non-Timeout arguments must satisfy `ConvertibleToFunctionValue` concept

Template Parameters

- **ReturnType** –Expected return type (default: void)
- **FunctionIdType** –Type of the function identifier (enum)
- **Args** –Types of function arguments (must be convertible to FunctionValue)

Parameters

- **function_id** –Function identifier
- **args** –Function arguments in order, last argument can be Timeout(ms)

Returns std::expected<ReturnType, std::string> Function result or error

template<typename **FunctionIdType**, typename ...**Args**>
static inline bool **call_function_async**(*FunctionIdType* function_id, *Args*&&... args)

Call a function asynchronously with variadic arguments.

```
// No arguments
call_function_async(FunctionId::GetValue);

// With arguments
call_function_async(FunctionId::Add, 1.0, 2.0);

// With handler
call_function_async(FunctionId::Add, 1.0, 2.0, [](FunctionResult &&result)
↪{
    // Handle result
});
```

Note: Arguments are packed into std::vector<FunctionValue> in the order provided

Note: All arguments (except optional FunctionResultHandler) must satisfy ConvertibleToFunctionValue concept

Note: This overload is not used when the first argument is FunctionParameterMap or boost::json::object

Template Parameters
- **FunctionIdType** –Type of the function identifier (enum)
- **Args** –Types of function arguments (must be convertible to FunctionValue, optionally with FunctionResultHandler at the end)

Parameters
- **function_id** –Function identifier
- **args** –Function arguments in order, optionally with FunctionResultHandler at the end

Returns true if the call was successfully submitted, false otherwise

template<typename **EventIdType**>
static inline *EventRegistry*::*SignalConnection* **subscribe_event**(*EventIdType* event_id, *EventRegistry*::*SignalSlot* slot)

Subscribe to an event with a raw SignalSlot.

```
auto conn = subscribe_event(EventId::ValueChanged,
    [](const std::string &event_name, const EventItemMap &items) {
        // Handle event with raw items directly
    });
```

Template Parameters **EventIdType** –Type of the event identifier (enum)

Parameters
- **event_id** –Event identifier
- **slot** –Signal slot function with signature: void(const std::string &event_name, const EventItemMap &event_items)

Returns *EventRegistry::SignalConnection* Connection object (scoped, automatically unsubscribes on destruction)

template<typename **T**>
static inline FunctionResult **to_function_result**(std::expected<*T*, std::string> &&result)

Helper function to convert std::expected to FunctionResult.

Template Parameters **T** –Return value type, can be void or any type convertible to FunctionValue

Parameters **result** –**[in]** std::expected object
Returns FunctionResult Converted result

template<auto **EventIdValue**>

class **EventMonitor**

Event monitor for monitoring and waiting for specific events.

```
// Create a monitor for WiFi GeneralEventHappened
WifiHelper::EventMonitor<WifiHelper::EventId::GeneralEventHappened>␣
↪monitor;
monitor.start();
// ... trigger some action ...
bool got_event = monitor.wait_for(std::vector<service::EventItem>{
↪"Connected", false}, 5000);
monitor.stop();
```

Template Parameters **EventIdValue** –The specific EventId enum value to monitor

Public Functions

inline bool **start**()

Start monitoring for events.
Returns true if successfully started monitoring, false if already monitoring or failed

inline void **stop**()

Stop monitoring for events.

inline void **clear**()

Clear all received events.

inline bool **wait_for**(const ReceivedItmes &expected_items, uint32_t timeout_ms)

Wait for an event containing specific items.
Parameters
- **expected_items** –The expected items to find
- **timeout_ms** –Maximum time to wait in milliseconds

Returns true if matching items were received, false on timeout

inline bool **wait_for_any**(uint32_t timeout_ms)

Wait for any event to be received within a timeout period.
Parameters **timeout_ms** –Maximum time to wait in milliseconds
Returns true if any event was received, false on timeout

inline size_t **get_count**() const

Get the number of received events.
Returns Number of events received since *start()* or last *clear()*

inline bool **has**(const ReceivedItmes &expected_items) const

Check if specific items have been received.
Parameters **expected_items** –The expected items to find
Returns true if matching items exists in received items

inline const std::vector<ReceivedItmes> &**get_all**() const

Get all received events (unfiltered)
Returns Copy of all received events

inline std::optional<ReceivedItmes> **get_last**() const

Get the last received event.

Returns std::optional containing the last received items, std::nullopt if no items were received

template<typename ...**Args**>
inline std::vector<std::tuple<*Args*...>> **get_all**() const

Get received events filtered by item types and extracted as tuples.

```
// Get events where first item is string and second is bool
auto events = monitor.get_all<std::string, bool>();
for (const auto& [str_val, bool_val] : events) {
    // use str_val and bool_val
}
```

Template Parameters **Args** – Expected types for each position in the event (bool, double, std::string, boost::json::object, boost::json::array, RawBuffer)
Returns Vector of tuples containing extracted values from matching events

template<typename ...**Args**>
inline std::optional<std::tuple<*Args*...>> **get_last**() const

Get the last received event filtered by item types and extracted as tuple.

```
// Get last event where first item is string and second is bool
auto last_event = monitor.get_last<std::string, bool>();
if (last_event.has_value()) {
    const auto &[event_str, is_unexpected] = last_event.value();
    // Or use std::get<0>/std::get<1>
    const auto &event_str = std::get<0>(last_event.value());
    bool is_unexpected = std::get<1>(last_event.value());
}

// Get last event with single boost::json::array item
auto last_scan = monitor.get_last<boost::json::array>();
if (last_scan.has_value()) {
    const auto &ap_infos_array = std::get<0>(last_scan.value());
}
```

Template Parameters **Args** – Expected types for each position in the event (bool, double, std::string, boost::json::object, boost::json::array, RawBuffer)
Returns std::optional containing tuple of extracted values if the last event matches, std::nullopt if no events or type mismatch

inline bool **is_running**() const

Check if currently running.
Returns true if actively running

Macros

BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_0(Derived, function_id, func_call)

Create a zero-parameter function handler based on Derived and FunctionId.

Example: *BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_0(MyService, MyService::FunctionId::GetVolume, function_get_volume())*

Parameters

- **Derived** – The derived class type
- **function_id** – FunctionId enum value
- **func_call** – Actual function call (e.g., function_get_volume())

BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_1(Derived, function_id, param_type, func_call)

Create a single-parameter function handler based on Derived and FunctionId.

Example: *BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_1(MyService, MyService::FunctionId::PlayUrl, std::string, function_play_url(PARAM))*

Parameters

- **Derived** –The derived class type
- **function_id** –FunctionId enum value
- **param_type** –Parameter C++ type (e.g., std::string, double)
- **func_call** –Function call, use PARAM as parameter placeholder

BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_2(Derived, function_id, param1_type, param2_type, func_call)

Create a two-parameter function handler based on Derived and FunctionId.

Example: BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_2(MyService, MyService::FunctionId::Add, double, double, function_add(PARAM1, PARAM2))

Parameters

- **Derived** –The derived class type
- **function_id** –FunctionId enum value
- **param1_type** –First parameter C++ type
- **param2_type** –Second parameter C++ type
- **func_call** –Function call, use PARAM1, PARAM2 as parameter placeholders

BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_3(Derived, function_id, param1_type, param2_type, param3_type, func_call)

Create a three-parameter function handler based on Derived and FunctionId.

Example: BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_3(MyService, MyService::FunctionId::SetConfig, std::string, int, bool, function_set_config(PARAM1, PARAM2, PARAM3))

Parameters

- **Derived** –The derived class type
- **function_id** –FunctionId enum value
- **param1_type** –First parameter C++ type
- **param2_type** –Second parameter C++ type
- **param3_type** –Third parameter C++ type
- **func_call** –Function call, use PARAM1, PARAM2, PARAM3 as parameter placeholders

Audio Helper

- Public header: `#include "brookesia/service_helper/audio.hpp"`

Overview This page documents the Audio helper Doxygen API: types, enums, methods, and macros.

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/audio.hpp

Classes

class **Audio** : public esp_brookesia::service::helper::*Base*<*Audio*>

Helper schema definitions for the audio service.

Public Types

enum class **PlayControlAction** : uint8_t

Playback control actions.

Values:

enumerator **Pause**

enumerator **Resume**

enumerator **Stop**

enum class **PlayState** : uint8_t

Playback states.

Values:

enumerator **Idle**

enumerator **Playing**

enumerator **Paused**

enum class **CodecFormat** : uint8_t

Codec related configurations.

Values:

enumerator **PCM**

enumerator **OPUS**

enumerator **G711A**

enumerator **Max**

enum class **FunctionId** : uint8_t

Audio service function identifiers.

Values:

enumerator **SetPlaybackConfig**

enumerator **SetEncoderStaticConfig**

enumerator **SetDecoderStaticConfig**

enumerator **SetAFE_Config**

enumerator **GetAFE_WakeWords**

enumerator **PauseAFE_WakeupEnd**

enumerator **ResumeAFE_WakeupEnd**

enumerator **PlayUrl**

enumerator **PlayUrls**

enumerator **PlayControl**

enumerator **SetVolume**

enumerator **GetVolume**

enumerator **SetMute**

enumerator **StartEncoder**

enumerator **StopEncoder**

enumerator **PauseEncoder**

enumerator **ResumeEncoder**

enumerator **StartDecoder**

enumerator **StopDecoder**

enumerator **FeedDecoderData**

enumerator **ResetData**

enumerator **Max**

enum class **EventId** : uint8_t

Audio service event identifiers.

Values:

enumerator **PlayStateChanged**

enumerator **AFE_EventHappened**

enumerator **EncoderDataReady**

enumerator **RecorderDataReady**

enumerator **Max**

enum class **FunctionSetPlaybackConfigParam** : uint8_t

Parameter keys for *FunctionId::SetPlaybackConfig*.

Values:

enumerator **Config**

enum class **FunctionSetEncoderStaticConfigParam** : uint8_t

Parameter keys for *FunctionId::SetEncoderStaticConfig*.

Values:

enumerator **Config**

enum class **FunctionSetDecoderStaticConfigParam** : uint8_t

Parameter keys for *FunctionId::SetDecoderStaticConfig*.

Values:

enumerator **Config**

enum class **FunctionSetAFE_ConfigParam** : uint8_t

Parameter keys for *FunctionId::SetAFE_Config*.

Values:

enumerator **Config**

enum class **FunctionPlayUrlParam** : uint8_t

Parameter keys for *FunctionId::PlayUrl*.

Values:

enumerator **Url**

enumerator **Config**

enum class **FunctionPlayUrlsParam** : uint8_t

Parameter keys for *FunctionId::PlayUrls*.

Values:

enumerator **Urls**

enumerator **Config**

enum class **FunctionPlayControlParam** : uint8_t

Parameter keys for *FunctionId::PlayControl*.

Values:

enumerator **Action**

enum class **FunctionSetVolumeParam** : uint8_t

Parameter keys for *FunctionId::SetVolume*.

Values:

enumerator **Volume**

enum class **FunctionSetMuteParam** : uint8_t

Parameter keys for *FunctionId::SetMute*.

Values:

enumerator **Enable**

enum class **FunctionStartEncoderParam** : uint8_t

Parameter keys for *FunctionId::StartEncoder*.

Values:

enumerator **Config**

enum class **FunctionStartDecoderParam** : uint8_t

Parameter keys for *FunctionId::StartDecoder*.

Values:

enumerator **Config**

enum class **FunctionFeedDecoderDataParam** : uint8_t

Parameter keys for *FunctionId::FeedDecoderData*.

Values:

enumerator **Data**

enum class **EventPlayStateChangedParam** : uint8_t

Item keys for *EventId::PlayStateChanged*.

Values:

enumerator **State**

enum class **EventAFE_EventHappenedParam** : uint8_t

Item keys for *EventId::AFE_EventHappened*.

Values:

enumerator **Event**

enum class **EventEncoderDataReadyParam** : uint8_t

Item keys for *EventId::EncoderDataReady*.

Values:

enumerator **Data**

enum class **EventRecorderDataReadyParam** : uint8_t

Item keys for *EventId::RecorderDataReady*.

Values:

enumerator **Data**

Public Static Functions

static inline constexpr std::string_view **get_name**()

Service name used by *ServiceManager*.

Returns std::string_view Stable service name.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get function schemas exported by audio service.

Returns std::span<const FunctionSchema> Static function schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas exported by audio service.

Returns std::span<const EventSchema> Static event schema span.

struct **AFE_Config**

struct **AFE_VAD_Config**

AFE related configurations.

Public Members

uint8_t **mode** = 4

VAD mode

uint32_t **min_speech_ms** = 64

Minimum speech duration

uint32_t **min_noise_ms** = 1000

Minimum noise duration

struct **AFE_WakeNetConfig**

Public Members

std::string **model_partition_label** = "model"

Wake model partition label

std::string **mn_language** = "cn"

Wake model language

uint32_t **start_timeout_ms** = 30000

Timeout before wake start

uint32_t **end_timeout_ms** = 10000

Timeout before wake end

struct **CodecGeneralConfig**

Public Members

uint8_t **channels**

Number of audio channels (1-4)

uint8_t **sample_bits**

Bit depth in bits (e.g., 8, 16, 24, 32)

uint32_t **sample_rate**

Sample rate in Hz (e.g., 8000, 16000, 24000, 32000, 44100, 48000)

uint8_t **frame_duration**

Frame duration in milliseconds

struct **DecoderDynamicConfig**

Public Members

CodecFormat **type**

Decoder codec type

CodecGeneralConfig **general**

Decoder common codec settings

struct **DecoderStaticConfig**

Decoder related configurations.

struct **EncoderDynamicConfig**

Public Members

CodecFormat **type**

Encoder codec type

CodecGeneralConfig **general**

Encoder common codec settings

std::variant<std::monostate, *EncoderExtraConfigOpus*> **extra** = std::monostate{}

Optional codec-specific settings

uint32_t **fetch_interval_ms** = 10

Encoder fetch interval in milliseconds

uint32_t **fetch_data_size** = 4096

Encoder fetch size in bytes

struct **EncoderExtraConfigOpus**

Public Members

bool **enable_vbr**

Enable Variable Bit Rate (VBR)

uint32_t **bitrate**

Bitrate in bps

struct **EncoderStaticConfig**

Encoder related configurations.

struct **MixerGainConfig**

Mixer related configurations.

Public Members

float **initial_gain**

Initial gain value

float **target_gain**

Target gain value

int **transition_time**

Transition duration in milliseconds

struct **PlaybackConfig**

Playback related configurations.

struct **PlayUrlConfig**

Runtime options for `PlayUrl` and `PlayUrls`.

Public Members

bool **interrupt** = true

Whether current playback can be interrupted

uint32_t **delay_ms** = 0

Delay before playback starts

uint32_t **loop_count** = 0

Number of extra loops

uint32_t **loop_interval_ms** = 0

Interval between loops

uint32_t **timeout_ms** = 0

Timeout for finishing playback

Emote Helper

- Public header: `#include "brookesia/service_helper/expression/emote.hpp"`

Overview This page documents the Expression/Emote helper Doxygen API: types, enums, methods, and macros.

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/expression/emote.hpp

Classes

class **ExpressionEmote** : public esp_brookesia::service::helper::*Base*<*ExpressionEmote*>

Helper schema definitions for the emote-expression service.

Public Types

enum class **EventMessageType**

Supported message categories rendered by the expression service.

Values:

enumerator **Idle**

enumerator **Speak**

enumerator **Listen**

enumerator **System**

enumerator **User**

enumerator **Battery**

enumerator **Max**

enum class **AssetSourceType**

Supported asset-source backends used when loading expression assets.

Values:

enumerator **Path**

enumerator **PartitionLabel**

enumerator **Max**

struct **AssetSource**

Description of one asset source used by the expression service.

Public Members

std::string **source**

Source identifier such as a path or partition label.

AssetSourceType **type**

How `source` should be interpreted.

bool **flag_enable_mmap** = false

Whether mmap-backed loading should be enabled.

struct **Config**

Runtime display and task configuration for the expression service.

Public Members

uint32_t **h_res** = 0

Horizontal resolution in pixels.

uint32_t **v_res** = 0

Vertical resolution in pixels.

size_t **buf_pixels** = 0
: Display buffer size in pixels.

uint32_t **fps** = 0
: Target render frame rate.

int **task_priority** = 0
: Render task priority.

int **task_stack** = 0
: Render task stack size in bytes.

int **task_affinity** = 0
: Core affinity for the render task.

bool **task_stack_in_ext** = false
: Whether the task stack should live in external memory.

bool **flag_swap_color_bytes** = false
: Whether output color bytes must be swapped.

bool **flag_double_buffer** = false
: Whether double buffering is enabled.

bool **flag_buff_dma** = false
: Whether display buffers must be DMA-capable.

bool **flag_buff_spiram** = false
: Whether display buffers may be allocated in SPIRAM.

struct **FlushReadyEventParam**
: Parameters delivered with the flush-ready event.

Public Members

int **x_start** = 0
: Left edge of the dirty region.

int **y_start** = 0
: Top edge of the dirty region.

int **x_end** = 0
: Right edge of the dirty region.

int **y_end** = 0
: Bottom edge of the dirty region.

const void ***data** = nullptr
: Pixel buffer for the dirty region.

NVS Helper

- Public header: `#include "brookesia/service_helper/nvs.hpp"`

Overview This page documents the NVS helper Doxygen API: types, enums, methods, and macros.

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/nvs.hpp

Classes

class **NVS** : public esp_brookesia::service::helper::*Base*<*NVS*>

Helper schema definitions for the *NVS* service.

Public Types

enum class **ValueType**

They are used as parameter and return types for functions and events. Users can access or modify these types via serialization and deserialization.

Values:

enumerator **Bool**

enumerator **Int**

enumerator **String**

enumerator **Max**

enum class **FunctionId**

NVS service function identifiers.

Values:

enumerator **List**

enumerator **Set**

enumerator **Get**

enumerator **Erase**

enumerator **Max**

enum class **EventId**

NVS service event identifiers.

Values:

enumerator **Max**

enum class **FunctionListParam**

Parameter keys for *FunctionId::List*.

Values:

enumerator **Nspace**

enum class **FunctionSetParam**

Parameter keys for *FunctionId::Set*.

Values:

enumerator **Nspace**

enumerator **KeyValuePairs**

enum class **FunctionGetParam**

Parameter keys for *FunctionId::Get*.

Values:

enumerator **Nspace**

enumerator **Keys**

enum class **FunctionEraseParam**

Parameter keys for *FunctionId::Erase*.

Values:

enumerator **Nspace**

enumerator **Keys**

using **Value** = std::variant<bool, int32_t, std::string>

Value type stored in *NVS* entries.

using **KeyValueMap** = std::map<std::string, *Value*>

Key-value mapping payload for batch set/get operations.

Public Static Functions

static inline constexpr std::string_view **get_name**()

Service name used by *ServiceManager*.

Returns std::string_view Stable service name.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get function schemas exported by *NVS* service.

Returns std::span<const FunctionSchema> Static function schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas exported by *NVS* service.

Returns std::span<const EventSchema> Empty span because *NVS* has no events.

template<typename **T**>
static inline std::expected<void, std::string> **save_key_value**(const std::string &nspace, const std::string &key, const *T* &value, uint32_t timeout_ms = *DEFAULT_TIMEOUT_MS*)

Save key-value pairs to the *NVS* namespace.

Direct Storage (No Serialization):

- `bool`: Stored directly as JSON boolean value (true/false)
- `int32_t`: Stored directly as JSON number (int64_t in JSON)
- Integer types with size <= 32 bits (int8_t, uint8_t, int16_t, uint16_t, char, short, etc.): Converted to int32_t and stored as JSON number for optimal performance

Serialized Storage:

- Integer types with size > 32 bits (int64_t, uint64_t, long long, etc.): Serialized to JSON string using BROOKESIA_DESCRIBE_JSON_SERIALIZE
- Floating point types (float, double): Serialized to JSON string using BROOKESIA_DESCRIBE_JSON_SERIALIZE
- String types (std::string, const char*): Serialized to JSON string using BROOKESIA_DESCRIBE_JSON_SERIALIZE
- Complex types (std::vector, std::map, custom structs, etc.): Serialized to JSON string using BROOKESIA_DESCRIBE_JSON_SERIALIZE

Note: The storage method depends on the type T:

Template Parameters **T** – The type of the value to save
Parameters

- **nspace** – The namespace of the key-value pairs to save
- **key** – The key of the key-value pair to save
- **value** – The value of the key-value pair to save
- **timeout_ms** – The timeout in milliseconds

Returns std::expected<void, std::string> The result of the operation

template<typename **T**>
static inline std::expected<*T*, std::string> **get_key_value**(const std::string &nspace, const std::string &key, uint32_t timeout_ms = *DEFAULT_TIMEOUT_MS*)

Get key-value pair from the *NVS* namespace.

Direct Retrieval (No Deserialization):

- `bool`: Retrieved directly from JSON boolean value
- `int32_t`: Retrieved directly from JSON number

- Integer types with size <= 32 bits (int8_t, uint8_t, int16_t, uint16_t, char, short, etc.): Retrieved directly from JSON number and converted to the target integer type

Deserialized Retrieval:

- Integer types with size > 32 bits (int64_t, uint64_t, long long, etc.): Retrieved directly from JSON string and deserialized to the target integer type
- Floating point types (float, double): Retrieved directly from JSON string and deserialized to the target floating point type
- String types (std::string): Retrieved directly from JSON string and deserialized to the target string type
- Complex types (std::vector, std::map, custom structs, etc.): Retrieved directly from JSON string and deserialized to the target complex type

---

Note: The retrieval method depends on the type T and matches the storage method used in *save_key_value()*:

Template Parameters **T** –The type of the value to retrieve
Parameters
- **nspace** –The namespace of the key-value pair to retrieve
- **key** –The key of the key-value pair to retrieve
- **timeout_ms** –The timeout in milliseconds

Returns std::expected<T, std::string> The retrieved value or error message

static inline std::expected<void, std::string> **erase_keys**(const std::string &nspace, const std::vector<std::string> &keys = {}, uint32_t timeout_ms = *DEFAULT_TIMEOUT_MS*)

Erase key-value pairs from the *NVS* namespace.

Parameters
- **nspace** –The namespace of the key-value pairs to erase
- **keys** –The keys of the key-value pairs to erase, optional. If not provided or empty, all key-value pairs in the namespace will be erased
- **timeout_ms** –The timeout in milliseconds

Returns std::expected<void, std::string> The result of the operation

Public Static Attributes

static constexpr uint32_t **DEFAULT_TIMEOUT_MS** = BROOKESIA_SERVICE_MANAGER_DEFAULT_CALL_FUNCTION_TIMEOUT_MS

Default timeout for synchronous *NVS* helper calls.

struct **EntryInfo**

Metadata for one entry in an *NVS* namespace.

Public Members

std::string **nspace**

Namespace name

std::string **key**

Entry key

ValueType **type**
Entry value type

SNTP Helper

- Public header: `#include "brookesia/service_helper/sntp.hpp"`

Overview This page documents the SNTP helper Doxygen API: types, enums, methods, and macros.

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/sntp.hpp

Classes

class **SNTP** : public esp_brookesia::service::helper::*Base*<*SNTP*>
Helper schema definitions for the *SNTP* service.

Public Types

enum class **FunctionId**
SNTP service function identifiers.
Values:

enumerator **SetServers**

enumerator **SetTimezone**

enumerator **Start**

enumerator **Stop**

enumerator **GetServers**

enumerator **GetTimezone**

enumerator **IsTimeSynced**

enumerator **ResetData**

enumerator **Max**

enum class **EventId**
SNTP service event identifiers.
Values:

enumerator **Max**

enum class **FunctionSetServersParam**

Parameter keys for *FunctionId::SetServers*.

Values:

enumerator **Servers**

enum class **FunctionSetTimezoneParam**

Parameter keys for *FunctionId::SetTimezone*.

Values:

enumerator **Timezone**

Public Static Functions

static inline constexpr std::string_view **get_name**()

Name of the *SNTP* service.

Returns std::string_view Stable service name.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get the function schemas exported by the *SNTP* service.

Returns std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get the event schemas exported by the *SNTP* service.

Returns std::span<const EventSchema> Empty span because *SNTP* exposes no events.

Video Helper

- Public header: `#include "brookesia/service_helper/video.hpp"`

Overview This page documents the Video helper Doxygen API: types, enums, methods, and macros.

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/video.hpp

Classes

class **Video**

Shared schema/type definitions for video encoder and decoder helper services.

Public Types

enum class **EncoderSinkFormat** : uint8_t

Encoder related configurations.

Values:

enumerator **H264**

enumerator **MJPEG**

enumerator **RGB565**

enumerator **RGB888**

enumerator **BGR888**

enumerator **YUV420**

enumerator **YUV422**

enumerator **O_UYY_E_VYY**

enumerator **Max**

enum class **DecoderSourceFormat** : uint8_t

Decoder related configurations.

Values:

enumerator **H264**

enumerator **MJPEG**

enumerator **Max**

enum class **EncoderFunctionId** : uint8_t

Video encoder function identifiers.

Values:

enumerator **Open**

enumerator **Close**

enumerator **Start**

enumerator **Stop**

enumerator **FetchFrame**

enumerator **Max**

enum class **EncoderEventId** : uint8_t

Video encoder event identifiers.

Values:

enumerator **StreamSinkFrameReady**

enumerator **FetchSinkFrameReady**

enumerator **Max**

enum class **DecoderFunctionId** : uint8_t

Video decoder function identifiers.

Values:

enumerator **Open**

enumerator **Close**

enumerator **Start**

enumerator **Stop**

enumerator **FeedFrame**

enumerator **Max**

enum class **DecoderEventId** : uint8_t

Video decoder event identifiers.

Values:

enumerator **SinkFrameReady**

enumerator **Max**

enum class **EncoderFunctionOpenParam** : uint8_t

Parameter keys for *EncoderFunctionId::Open*.

Values:

enumerator **Config**

enum class **EncoderFunctionFetchFrameParam** : uint8_t

Parameter keys for *EncoderFunctionId::FetchFrame*.

Values:

enumerator **SinkIndex**

enum class **DecoderFunctionOpenParam** : uint8_t

Parameter keys for *DecoderFunctionId::Open*.

Values:

enumerator **Config**

enum class **DecoderFunctionFeedFrameParam** : uint8_t

Parameter keys for *DecoderFunctionId::FeedFrame*.

Values:

enumerator **Frame**

enum class **EncoderEventStreamSinkFrameReadyParam** : uint8_t

Item keys for *EncoderEventId::StreamSinkFrameReady*.

Values:

enumerator **SinkIndex**

enumerator **SinkInfo**

enumerator **Frame**

enum class **EncoderEventFetchSinkFrameReadyParam** : uint8_t

Item keys for *EncoderEventId::FetchSinkFrameReady*.

Values:

enumerator **SinkIndex**

enumerator **SinkInfo**

enumerator **Frame**

enum class **DecoderEventSinkFrameReadyParam** : uint8_t

Item keys for *DecoderEventId::SinkFrameReady*.

Values:

enumerator **Width**

enumerator **Height**

enumerator **Frame**

Public Static Functions

static inline std::span<const FunctionSchema> **get_encoder_function_schemas**()

Get all encoder function schemas.

Returns std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_encoder_event_schemas**()

Get all encoder event schemas.

Returns std::span<const EventSchema> Static schema span.

static inline std::span<const FunctionSchema> **get_decoder_function_schemas**()

Get all decoder function schemas.

Returns std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_decoder_event_schemas**()

Get all decoder event schemas.

Returns std::span<const EventSchema> Static schema span.

Public Static Attributes

static constexpr std::string_view **ENCODER_NAME_PREFIX** = "VideoEncoder"

Prefix used to build encoder helper service names.

static constexpr std::string_view **DECODER_NAME_PREFIX** = "VideoDecoder"

Prefix used to build decoder helper service names.

struct **DecoderConfig**

Decoder open configuration.

Public Members

uint16_t **width**

Decode frame width

uint16_t **height**

Decode frame height

DecoderSourceFormat **source_format**

Decoder input format

DecoderSinkFormat **sink_format**

Decoder output format

bool **enable_stream_mode**

Whether decoder works in stream mode

bool **enable_hw_acceleration**
Whether hardware acceleration is enabled

struct **EncoderConfig**
Encoder open configuration.

Public Members

std::vector<*EncoderSinkInfo*> **sinks**
Output sink list

bool **enable_stream_mode**
Whether encoder works in stream push mode

struct **EncoderSinkInfo**
One encoder sink stream description.

Public Members

EncoderSinkFormat **format**
Output sink format

uint16_t **width**
Output width

uint16_t **height**
Output height

uint8_t **fps**
Output frames per second

template<int **Id**>

class **VideoEncoder** : public esp_brookesia::service::helper::*Base*<*VideoEncoder*<*Id*>>

Public Types

using **FunctionId** = *Video*::*EncoderFunctionId*
Re-exported function id enum for encoder service instance.

using **EventId** = *Video*::*EncoderEventId*
Re-exported event id enum for encoder service instance.

Public Static Functions

static inline std::string_view **get_name**()
Get service name of this encoder instance.

Returns std::string_view Service name in format `VideoEncoder<Id>`.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get function schemas for this encoder instance.

Returns std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas for this encoder instance.

Returns std::span<const EventSchema> Static schema span.

template<int **Id**>

class **VideoDecoder** : public esp_brookesia::service::helper::*Base*<*VideoDecoder*<*Id*>>

Public Types

using **FunctionId** = *Video*::*DecoderFunctionId*

Re-exported function id enum for decoder service instance.

using **EventId** = *Video*::*DecoderEventId*

Re-exported event id enum for decoder service instance.

Public Static Functions

static inline std::string_view **get_name**()

Get service name of this decoder instance.

Returns std::string_view Service name in format `VideoDecoder<Id>`.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get function schemas for this decoder instance.

Returns std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas for this decoder instance.

Returns std::span<const EventSchema> Static schema span.

Wi-Fi Helper

- Public header: `#include "brookesia/service_helper/wifi.hpp"`

Overview This page documents the Wi-Fi helper Doxygen API: types, enums, methods, and macros.

API reference

Header File

- service/brookesia_service_helper/include/brookesia/service_helper/wifi.hpp

Classes

class **Wifi** : public esp_brookesia::service::helper::*Base*<*Wifi*>

Helper schema definitions for the Wi-Fi service.

Public Types

enum class **GeneralAction**

General Wi-Fi control actions.

Values:

enumerator **Init**

Initialize Wi-Fi resources.

enumerator **Deinit**

Deinitialize Wi-Fi resources.

enumerator **Start**

Start Wi-Fi subsystem.

enumerator **Stop**

Stop Wi-Fi subsystem.

enumerator **Connect**

Connect to configured AP.

enumerator **Disconnect**

Disconnect from current AP.

enumerator **Max**

Sentinel value.

enum class **GeneralEvent**

General Wi-Fi lifecycle events.

Values:

enumerator **Deinited**

Wi-Fi has been deinitialized.

enumerator **Inited**

Wi-Fi has been initialized.

enumerator **Stopped**

Wi-Fi has been stopped.

enumerator **Started**

Wi-Fi has started.

enumerator **Disconnected**

Wi-Fi disconnected from AP.

enumerator **Connected**

Wi-Fi connected to AP.

enumerator **Max**

Sentinel value.

enum class **GeneralState**

General state for WiFi state machine.

Stable states: Idle, Inited, Started, Connected Transient states: Initing, Deiniting, Starting, Stopping, Connecting, Disconnecting

Values:

enumerator **Idle**

Stable: Wi-Fi is not initialized.

enumerator **Initing**

Transient: Wi-Fi is initializing.

enumerator **Inited**

Stable: Wi-Fi initialized but not started.

enumerator **Deiniting**

Transient: Wi-Fi is deinitializing.

enumerator **Starting**

Transient: Wi-Fi is starting.

enumerator **Started**

Stable: Wi-Fi started but not connected.

enumerator **Stopping**

Transient: Wi-Fi is stopping.

enumerator **Connecting**

Transient: Wi-Fi is connecting.

enumerator **Connected**

Stable: Wi-Fi is connected.

enumerator **Disconnecting**

Transient: Wi-Fi is disconnecting.

enumerator **Max**

Sentinel value.

enum class **ScanApSignalLevel**

Signal-strength bucket for a scanned AP.

Values:

enumerator **LEVEL_0**

RSSI <= -81 dBm.

enumerator **LEVEL_1**

RSSI in [-80, -71] dBm.

enumerator **LEVEL_2**

RSSI in [-70, -61] dBm.

enumerator **LEVEL_3**

RSSI in [-60, -51] dBm.

enumerator **LEVEL_4**

RSSI >= -50 dBm.

enum class **SoftApEvent**

SoftAP lifecycle events.

Values:

enumerator **Started**

SoftAP started.

enumerator **Stopped**

SoftAP stopped.

enumerator **Max**

Sentinel value.

enum class **FunctionId**

Wi-Fi service function identifiers.

Values:

enumerator **TriggerGeneralAction**

enumerator **GetGeneralState**

enumerator **SetConnectAp**

enumerator **GetConnectAp**

enumerator **GetConnectedAps**

enumerator **SetScanParams**

enumerator **TriggerScanStart**

enumerator **TriggerScanStop**

enumerator **SetSoftApParams**

enumerator **GetSoftApParams**

enumerator **TriggerSoftApStart**

enumerator **TriggerSoftApStop**

enumerator **TriggerSoftApProvisionStart**

enumerator **TriggerSoftApProvisionStop**

enumerator **ResetData**

enumerator **Max**

enum class **EventId**

Wi-Fi service event identifiers.

Values:

enumerator **GeneralActionTriggered**

enumerator **GeneralEventHappened**

enumerator **ScanStateChanged**

enumerator **ScanApInfosUpdated**

enumerator **SoftApEventHappened**

enumerator **Max**

enum class **FunctionTriggerGeneralActionParam**

Parameter keys for *FunctionId::TriggerGeneralAction*.

Values:

enumerator **Action**

enum class **FunctionSetConnectApParam**

Parameter keys for *FunctionId::SetConnectAp*.

Values:

enumerator **SSID**

enumerator **Password**

enum class **FunctionSetScanParamsParam**

Parameter keys for *FunctionId::SetScanParams*.

Values:

enumerator **Param**

enum class **FunctionSetSoftApParamsParam**

Parameter keys for *FunctionId::SetSoftApParams*.

Values:

enumerator **Param**

enum class **EventGeneralActionTriggeredParam**

Item keys for *EventId::GeneralActionTriggered*.

Values:

enumerator **Action**

enum class **EventGeneralEventHappenedParam**

Item keys for *EventId::GeneralEventHappened*.

Values:

enumerator **Event**

enumerator **IsUnexpected**

enum class **EventScanStateChangedParam**

Item keys for *EventId::ScanStateChanged*.

Values:

enumerator **IsRunning**

enum class **EventScanApInfosUpdatedParam**

Item keys for *EventId::ScanApInfosUpdated*.

Values:

enumerator **ApInfos**

enum class **EventSoftApEventHappenedParam**

Item keys for *EventId::SoftApEventHappened*.

Values:

enumerator **Event**

Public Static Functions

static inline constexpr std::string_view **get_name**()

Get helper contract name.

Returns Constant service name string.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get all function schemas exposed by this helper.

Returns Read-only span of function schemas.

static inline std::span<const EventSchema> **get_event_schemas**()

Get all event schemas exposed by this helper.

Returns Read-only span of event schemas.

struct **ConnectApInfo**

Target AP connection info.

Public Members

std::string **ssid**

AP SSID.

std::string **password**

AP password.

bool **is_connectable** = true

Whether this AP is considered connectable.

struct **ScanApInfo**

One scanned AP entry.

Public Members

std::string **ssid**

AP SSID.

bool **is_locked**

Whether AP authentication is required.

int **rssi**

Signal strength in dBm.

ScanApSignalLevel **signal_level**

Bucketed signal level derived from RSSI.

uint8_t **channel**

RF channel.

Public Static Functions

static inline *ScanApSignalLevel* **get_signal_level**(int rssi)

Convert RSSI to a signal-level bucket.

Parameters **rssi** –**[in]** RSSI value in dBm.

Returns Corresponding `ScanApSignalLevel`.

struct **ScanParams**

Parameters for periodic AP scanning.

Public Members

size_t **ap_count** = 20

Maximum number of AP entries to keep.

uint32_t **interval_ms** = 10000

Scan interval in milliseconds.

uint32_t **timeout_ms** = 60000

Total scan timeout in milliseconds.

struct **SoftApParams**

SoftAP runtime configuration.

Public Members

std::string **ssid** = ""

SoftAP SSID.

std::string **password** = ""

SoftAP password.

uint8_t **max_connection** = 4

Maximum station connections.

std::optional<uint8_t> **channel** = std::nullopt

Optional fixed channel.

4.2 General services

4.2.1 NVS

- Component registry: espressif/brookesia_service_nvs
- Helper header: `#include "brookesia/service_helper/nvs.hpp"`
- Helper class: `esp_brookesia::service::helper::NVS`

Overview

brookesia_service_nvs provides NVS (non-volatile storage) for the ESP-Brookesia ecosystem:

- **Namespaces**: Key–value storage partitioned by namespace.
- **Types**: Boolean, 32-bit signed integer, and UTF-8 string.
- **Persistence**: Data survives power cycles in the NVS partition.
- **Thread safety**: Optional *TaskScheduler*–based async execution.
- **Queries**: List keys in a namespace or read specific keys.

Features

Namespaces

- **Default**: If omitted, namespace `"storage"` is used.
- **Multiple**: Create several namespaces for different data.
- **Isolation**: Namespaces are independent.

Supported types

- **Bool**: `true` / `false`.
- **Int**: 32-bit signed integer.
- **String**: UTF-8 string.

Note: Beyond these primitives, NVS Helper templates *`esp_brookesia::service::helper::NVS::save_key_value(`* and *`esp_brookesia::service::helper::NVS::get_key_value()`* can store and retrieve arbitrary types.

Core capabilities

- List key metadata in a namespace.
- Set multiple key–value pairs in batch.
- Read one key or all pairs in a namespace.
- Delete a key or clear a namespace.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/service_helper/nvs.hpp\"`
- Helper class: `esp_brookesia::service::helper::NVS`

Service Interfaces

Functions

`List`

Description List key-value entries in an NVS namespace. Return type: JSON array<object>. Example: [{ “nspace” :” storage” ,” key” :” key1” ,” type” :” String” },{ “nspace” :” storage” ,” key” :” key2” ,” type” :” Int” }]

Execution

- Requires scheduler: Required

Parameters

- `Nspace`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `"storage"`
  - **Description**: Namespace to list (optional). Uses the default namespace when omitted.

Schema JSON

CLI Command

```
svc_call NVS List {"Nspace":null}
```

Set

Description Set key-value pairs in an NVS namespace.

Execution

- Requires scheduler: Required

Parameters

- `Nspace`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `"storage"`
  - **Description**: Namespace to write (optional). Uses the default namespace when omitted or empty.
- `KeyValuePairs`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Key-value pairs as a JSON object. Allowed value types: [“Bool” ,” Int” ,” String”]. Example: { “key1” :” value1” ,” key2” :2,” key3” :true}

Schema JSON

CLI Command

```
svc_call NVS Set {"Nspace":null,"KeyValuePairs":null}
```

Get

Description Get key-value pairs by keys from an NVS namespace. Return type: JSON object. Example: { “key1” :” value1” ,” key2” :2,” key3” :true}

Execution

- Requires scheduler: Required

Parameters

- `Nspace`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `"storage"`
  - **Description**: Namespace to read (optional). Uses the default namespace when omitted.
- `Keys`
  - **Type**: `Array`
  - **Required**: optional
  - **Default**: `[]`
  - **Description**: Keys to read as JSON array<string> (optional). Returns all pairs when omitted. Example: [“key1” ,” key2” ,” key3”]

Schema JSON

CLI Command

```
svc_call NVS Get {"Nspace":null,"Keys":null}
```

Erase

Description Erase key-value pairs from an NVS namespace.

Execution

- Requires scheduler: Required

Parameters

- `Nspace`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `"storage"`
  - **Description**: Namespace to erase (optional). Uses the default namespace when omitted.
- `Keys`
  - **Type**: `Array`
  - **Required**: optional
  - **Default**: `[]`
  - **Description**: Keys to erase as JSON array<string> (optional). Erases all pairs when omitted or empty. Example: [“key1” ,” key2” ,” key3”]

Schema JSON

CLI Command

```
svc_call NVS Erase {"Nspace":null,"Keys":null}
```

Events This contract does not publish standard event schemas.

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

4.2.2 SNTP

- Component registry: espressif/brookesia_service_sntp
- Helper header: `#include "brookesia/service_helper/sntp.hpp"`
- Helper class: `esp_brookesia::service::helper::SNTP`

Overview

brookesia_service_sntp provides SNTP (Simple Network Time Protocol) for the ESP-Brookesia ecosystem:

- **NTP servers**: Multiple servers; picks an available one.
- **Time zone**: System time zone with offset applied.
- **Auto sync**: Starts when the network is up.
- **Status**: Query sync state, server list, and zone.
- **Persistence**: Optional *brookesia_service_nvs* for servers and zone.

Features

NTP servers

- Default: `"pool.ntp.org"`.
- Multiple servers with automatic selection.
- Query the configured list.

Time zone

- Default: CST-8 (China, UTC+8).
- Standard zone strings (`UTC`, `CST-8`, `EST-5`, ⋯).
- Applied to system time when set.

Core operations

- **Set NTP servers**: Configure one or more NTP servers.
- **Set time zone**: Configure the system time zone.
- **Start / stop service**: Start or stop SNTP and time synchronization.
- **Get server list**: Read the configured NTP server list.
- **Get time zone**: Read the configured time zone.
- **Check sync status**: Check whether system time is synchronized with NTP.
- **Reset data**: Reset all configuration to defaults.

Auto management

- Load from NVS on start.
- Save after changes.
- Detect network and start sync when available.
- Monitor sync status and flags.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/service_helper/sntp.hpp\"`
- Helper class: `esp_brookesia::service::helper::SNTP`

Service Interfaces

Functions

SetServers

Description Set NTP servers.

Execution

- Requires scheduler: Not required

Parameters

- Servers
  - **Type**: Array
  - **Required**: required
  - **Description**: NTP servers as JSON array<string>. Example: [“pool.ntp.org” ,” cn.pool.ntp.org”]

Schema JSON

CLI Command

```
svc_call SNTP SetServers {"Servers":null}
```

SetTimezone

Description Set timezone.

Execution

- Requires scheduler: Not required

Parameters

- Timezone
  - **Type**: String
  - **Required**: required
  - **Description**: Timezone string.

Schema JSON

CLI Command

```
svc_call SNTP SetTimezone {"Timezone":null}
```

Start

Description Start SNTP service.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call SNTP Start
```

Stop

Description Stop SNTP service.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call SNTP Stop
```

GetServers

Description Get NTP servers. Return type: JSON array<string>. Example: [“pool.ntp.org” ,” cn.pool.ntp.org”]

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call SNTP GetServers
```

GetTimezone

Description Get timezone. Return type: string. Example: "CST-8"

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call SNTP GetTimezone
```

IsTimeSynced

Description Check whether time is synced. Return type: boolean. Example: true

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call SNTP IsTimeSynced
```

ResetData

Description Reset NTP servers, timezone, and sync status.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call SNTP ResetData
```

Events This contract does not publish standard event schemas.

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

4.2.3 Wi-Fi

- Component registry: espressif/brookesia_service_wifi
- Helper header: `#include "brookesia/service_helper/wifi.hpp"`
- Helper class: `esp_brookesia::service::helper::Wifi`

Overview

brookesia_service_wifi manages Wi-Fi connectivity:

- **State machine**: Init, start, connect, and related lifecycle states.
- **Auto reconnect**: Reconnect to known APs after drops.
- **Scanning**: Periodic scans and discovery of connectable APs.
- **Connection management**: Target AP and connected-AP lists with history.
- **Events**: Rich notifications for state changes.
- **Persistence**: Optional *brookesia_service_nvs* for credentials and parameters.

Features

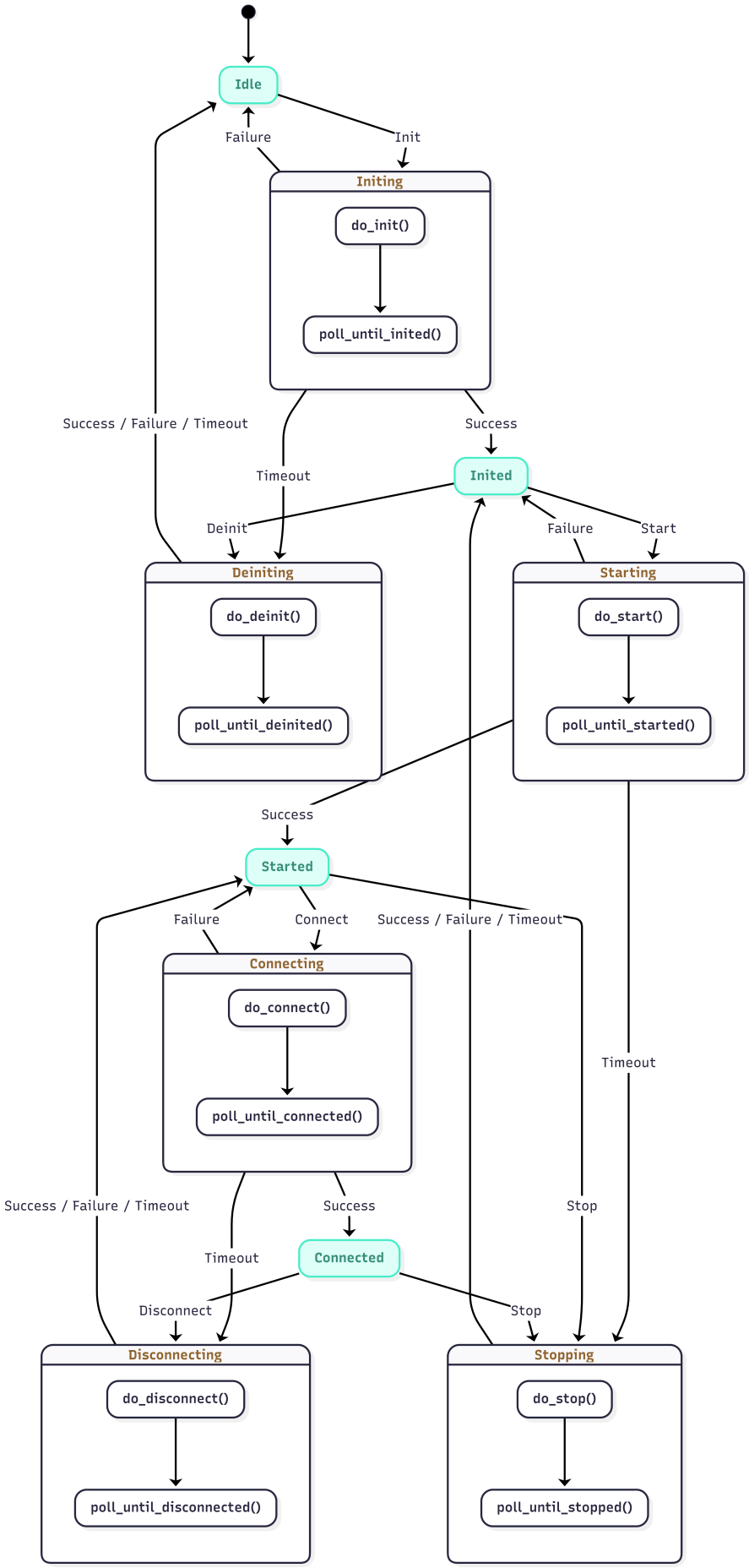
State machine The Wi-Fi service manages lifecycle states in a state machine for safe, consistent transitions. The state machine has four core states:

State	Description
`Idle`	Wi-Fi not initialized (initial state).
`Inited`	Initialized but not started; parameters can be configured.
`Started`	Started; scanning or waiting to connect.
`Connected`	Associated with an AP and ready for traffic.

Transitions use **actions**:

- **Forward**: Idle → Inited (Init) → Started (Start) → Connected (Connect)
- **Disconnect**: Connected → Started (Disconnect)
- **Stop**: Started / Connected → Inited (Stop)
- **Deinit**: Inited → Idle (Deinit)

State transition diagram:


Idle
Failure
Init
Initing
do_init()
poll_until_inited()
Success / Failure / Timeout
Success
Timeout
Inited
Deinit
Failure
Start
Deiniting
do_deinit()
poll_until_deinited()
Starting
do_start()
poll_until_started()
Success
Started
Failure
Connect
Success / Failure / Timeout
Connecting
do_connect()
poll_until_connected()
Timeout
Success / Failure / Timeout
Success
Stop
Timeout
Connected
Disconnect
Stop
Disconnecting
do_disconnect()
poll_until_disconnected()
Stopping
do_stop()
poll_until_stopped()

Auto reconnect

- Connect to historical APs after start.
- Reconnect after unexpected disconnects.
- Auto-connect when a target or historical AP appears during scan.

Scanning

- Configurable interval and timeout.
- Events with SSID, signal level, security, etc.

SoftAP

- SSID, password, max clients, channel.
- Optional automatic best-channel selection.
- SoftAP provisioning mode.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/service_helper/wifi.hpp\"`
- Helper class: `esp_brookesia::service::helper::Wifi`

Service Interfaces

Functions

`TriggerGeneralAction`

Description Trigger a Wi-Fi general action.

Execution

- Requires scheduler: Required

Parameters

- `Action`
  - **Type**: `String`
  - **Required**: required
  - **Description**: General action. Allowed values: [Init, Deinit, Start, Stop, Connect, Disconnect]

Schema JSON

CLI Command

```
svc_call Wifi TriggerGeneralAction {"Action":null}
```

`GetGeneralState`

Description Get current general state. Return type: string. Allowed values: [Idle, Initing, Inited, Deiniting, Starting, Started, Stopping, Connecting, Connected, Disconnecting]. Example: "Connected"

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi GetGeneralState
```

SetConnectAp

Description Set target AP credentials.

Execution

- Requires scheduler: Required

Parameters

- `SSID`
  - **Type**: `String`
  - **Required**: required
  - **Description**: AP SSID.
- `Password`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `""`
  - **Description**: AP password (optional).

Schema JSON

CLI Command

```
svc_call Wifi SetConnectAp {"SSID":null,"Password":null}
```

GetConnectAp

Description Get target AP. Return type: JSON object. Example: { “ssid” :” ssid1” ,” password” :” password1” ,” is_connectable” :true}

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi GetConnectAp
```

GetConnectedAps

Description Get connected AP list. Return type: JSON array<object>. Example: [{ “ssid” :” ssid1” ,” password” :” password1” ,” is_connectable” :true},{ “ssid” :” ssid2” ,” password” :” password2” ,” is_connectable” :true}]

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi GetConnectedAps
```

SetScanParams

Description Set scan parameters.

Execution

- Requires scheduler: Required

Parameters

- `Param`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Scan parameters as a JSON object. Example: { “ap_count” :20,” interval_ms” :10000,” timeout_ms” :60000}

Schema JSON

CLI Command

```
svc_call Wifi SetScanParams {"Param":null}
```

TriggerScanStart

Description Start Wi-Fi scan.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi TriggerScanStart
```

`TriggerScanStop`

Description Stop Wi-Fi scan.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi TriggerScanStop
```

`SetSoftApParams`

Description Set SoftAP parameters.

Execution

- Requires scheduler: Required

Parameters

- `Param`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: SoftAP parameters as a JSON object. Example: { “ssid” :” ssid” ,” password” :” password” ,” max_connection” :4,” channel” :1}

Schema JSON

CLI Command

```
svc_call Wifi SetSoftApParams {"Param":null}
```

GetSoftApParams

Description Get SoftAP parameters. Return type: JSON object. Example: { “ssid” :”” ,” password” :”” ,” max_connection” :4,” channel” :null}

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi GetSoftApParams
```

TriggerSoftApStart

Description Start SoftAP.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi TriggerSoftApStart
```

TriggerSoftApStop

Description Stop SoftAP.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi TriggerSoftApStop
```

`TriggerSoftApProvisionStart`

Description Start SoftAP provisioning.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi TriggerSoftApProvisionStart
```

`TriggerSoftApProvisionStop`

Description Stop SoftAP provisioning.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi TriggerSoftApProvisionStop
```

`ResetData`

Description Reset Wi-Fi data, including target AP, scan parameters, and connected APs. Also clears NVS data.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Wifi ResetData
```

Events

`GeneralActionTriggered`

Description Emitted when a general action is triggered.

Execution

- Requires scheduler: Required

Items

- `Action`
  - **Type**: `String`
  - **Description**: General action. Allowed values: [Init, Deinit, Start, Stop, Connect, Disconnect]

Schema JSON

CLI Command

```
svc_subscribe Wifi GeneralActionTriggered
```

`GeneralEventHappened`

Description Emitted when a general event occurs.

Execution

- Requires scheduler: Required

Items

- `Event`
  - **Type**: `String`
  - **Description**: General event. Allowed values: [Deinited, Inited, Stopped, Started, Disconnected, Connected]
- `IsUnexpected`
  - **Type**: `Boolean`
  - **Description**: Whether the event was unexpected.

Schema JSON

CLI Command

```
svc_subscribe Wifi GeneralEventHappened
```

ScanStateChanged

Description Emitted when scan state changes.

Execution

- Requires scheduler: Required

Items

- `IsRunning`
  - **Type**: `Boolean`
  - **Description**: Whether scanning is running.

Schema JSON

CLI Command

```
svc_subscribe Wifi ScanStateChanged
```

ScanApInfosUpdated

Description Emitted when scan AP list is updated.

Execution

- Requires scheduler: Required

Items

- `ApInfos`
  - **Type**: `Array`
  - **Description**: Scanned AP list as a JSON array<object>. Example: [{ “ssid” :” ssid1” ,” password” :” password1” ,” is_connectable” :true},{ “ssid” :” ssid2” ,” password” :” password2” ,” is_connectable” :true}]

Schema JSON

CLI Command

```
svc_subscribe Wifi ScanApInfosUpdated
```

SoftApEventHappened

Description Emitted when a SoftAP event occurs.

Execution

- Requires scheduler: Required

Items

- `Event`
  - **Type**: `String`
  - **Description**: SoftAP event. Allowed values: [Started, Stopped]

Schema JSON

CLI Command

```
svc_subscribe Wifi SoftApEventHappened
```

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

4.2.4 Audio

- Component registry: espressif/brookesia_service_audio
- Helper header: `#include "brookesia/service_helper/audio.hpp"`
- Helper class: `esp_brookesia::service::helper::Audio`

Overview

brookesia_service_audio provides:

- **Playback**: Stream from URL; pause, resume, stop.
- **Codecs**: PCM, OPUS, G711A encode/decode.
- **Volume**: Set and query volume.
- **Playback state**: Idle / playing / paused with events.
- **Encoder**: Start/stop/configure; configurable read size.
- **Decoder**: Start/stop and feed compressed data; streaming decode.
- **Persistence**: Optional *brookesia_service_nvs* for volume and related settings.

Features

Codec formats The audio service supports the following codec formats:

Format	Encode	Decode	Notes
PCM	Yes	Yes	Lossless
OPUS	Yes	Yes	VBR and fixed bitrate
G711A	Yes	Yes	Telephony quality

Playback

- URL playback.
- Pause, resume, stop.
- State events for UI sync.

Encoder configuration

- Codecs: PCM, OPUS, G711A.
- Channels: 1–4.
- Bits per sample: 8, 16, 24, 32.
- Sample rates: 8000, 16000, 24000, 32000, 44100, 48000 Hz.
- Frame duration (ms).
- OPUS: VBR and bitrate.

Decoder configuration

- **Codecs**: PCM, OPUS, G711A.
- **Channels**: 1–4.
- **Bits per sample**: 8, 16, 24, 32.
- **Sample rates**: 8000, 16000, 24000, 32000, 44100, 48000 Hz.
- **Frame duration (ms)**.

Events

- Playback state changes (Idle, Playing, Paused).
- Encoder events.
- Encoded data ready.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/service_helper/audio.hpp\"`
- Helper class: `esp_brookesia::service::helper::Audio`

Service Interfaces

Functions

`SetPlaybackConfig`

Description Set playback config. Call before starting the service.

Execution

- Requires scheduler: Not required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Playback config. Example: { “player_task” :{ “name” :” Thread” ,” core_id” :0,” priority” :5,” stack_size” :4096,” stack_in_ext” :true},” mixer_gain” :{ “initial_gain” :6.000000238418579E-1,” target_gain” :1E0,” transition_time” :1500}}

Schema JSON

CLI Command

```
svc_call Audio SetPlaybackConfig {"Config":null}
```

SetEncoderStaticConfig

Description Set encoder static config. Call before starting the service.

Execution

- Requires scheduler: Not required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Encoder static config. Example: { “recorder_task” :{ “name” :” Thread” ,” core_id” :0,” priority” :10,” stack_size” :4096,” stack_in_ext” :true},” fetcher_task” :{ “name” :” EncoderFetcher” ,” core_id” :1,” priority” :12,” stack_size” :6144,” stack_in_ext” :true}}

Schema JSON

CLI Command

```
svc_call Audio SetEncoderStaticConfig {"Config":null}
```

SetDecoderStaticConfig

Description Set decoder static config. Call before starting the service.

Execution

- Requires scheduler: Not required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Decoder static config. Example: {“feeder_task”:{“name”:”Thread”,”core_id”:1,”priority”:5,”stack_size”:4096,”stack_in_ext”:true},”mixer_gain”:{“initial_gain”:8.999999761581421E-1,” target_gain” :1E0,” transition_time” :1500}}

Schema JSON

CLI Command

```
svc_call Audio SetDecoderStaticConfig {"Config":null}
```

SetAFE_Config

Description Set AFE config. Call before starting the service.

Execution

- Requires scheduler: Not required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: AFE config. Example: { “feeder_task” :{ “name” :” Thread” ,” core_id” :1,” priority” :5,” stack_size” :40960,” stack_in_ext” :true},” fetcher_task” :{ “name” :” Thread” ,” core_id” :0,” priority” :5,” stack_size” :6144,” stack_in_ext” :true},” vad” :null,” wakenet” :null}

Schema JSON

CLI Command

```
svc_call Audio SetAFE_Config {"Config":null}
```

GetAFE_WakeWords

Description Get AFE wake words. Return type: JSON array<string>. Example: [“ni hao xiao zhi” ,” hello brookesia”]

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio GetAFE_WakeWords
```

PauseAFE_WakeupEnd

Description Pause AFE wakeup-end task.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio PauseAFE_WakeupEnd
```

`ResumeAFE_WakeupEnd`

Description Resume AFE wakeup-end task.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio ResumeAFE_WakeupEnd
```

`PlayUrl`

Description Play audio from a URL. Supports loop and interrupt playback.

Execution

- Requires scheduler: Required

Parameters

- `Url`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Audio URL, for example: “file://spiffs/example.mp3” .
- `Config`
  - **Type**: `Object`
  - **Required**: optional
  - **Default**: `{"interrupt":true,"delay_ms":0,"loop_count":0,"loop_interval_ms":0,"timeout_ms":0}`
  - **Description**: Playback config. Example: { “interrupt” :true,” delay_ms” :0,” loop_count” :0,” loop_interval_ms” :0,” timeout_ms” :0}

Schema JSON

CLI Command

```
svc_call Audio PlayUrl {"Url":null,"Config":null}
```

PlayUrls

Description Play audio from multiple URLs. Supports loop and interrupt playback.

Execution

- Requires scheduler: Required

Parameters

- `Urls`
  - **Type**: `Array`
  - **Required**: required
  - **Description**: Audio URL list. Example: [" file://spiffs/example1.mp3" ," file://spiffs/example2.mp3"]
- `Config`
  - **Type**: `Object`
  - **Required**: optional
  - **Default**: `{"interrupt":true,"delay_ms":0,"loop_count":0, "loop_interval_ms":0,"timeout_ms":0}`
  - **Description**: Playback config. Example: { "interrupt" :true," delay_ms" :0," loop_count" :0," loop_interval_ms" :0," timeout_ms" :0}

Schema JSON

CLI Command

```
svc_call Audio PlayUrls {"Urls":null,"Config":null}
```

PlayControl

Description Control audio playback.

Execution

- Requires scheduler: Required

Parameters

- `Action`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Playback action. Allowed values: [Pause, Resume, Stop]

Schema JSON

CLI Command

```
svc_call Audio PlayControl {"Action":null}
```

SetVolume

Description

Set playback volume.

Execution

- Requires scheduler: Required

Parameters

- `Volume`
  - **Type**: `Number`
  - **Required**: required
  - **Description**: Volume in range [0, 100].

Schema JSON

CLI Command

```
svc_call Audio SetVolume {"Volume":null}
```

GetVolume

Description

Get playback volume. Return type: number. Example: 70

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio GetVolume
```

SetMute

Description

Set playback mute.

Execution

- Requires scheduler: Required

Parameters

- `Enable`
  - **Type**: `Boolean`
  - **Required**: required
  - **Description**: Enable mute.

Schema JSON

CLI Command

```
svc_call Audio SetMute {"Enable":null}
```

`StartEncoder`

Description Start audio encoder.

Execution

- Requires scheduler: Required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Audio encoder config. Example: { “type” :” PCM” ,” general” :{ “channels” :0,” sample_bits” :0,” sample_rate” :0,” frame_duration” :0},” extra” :null,” fetch_interval_ms” :10,” fetch_data_size” :4096}

Schema JSON

CLI Command

```
svc_call Audio StartEncoder {"Config":null}
```

`StopEncoder`

Description Stop audio encoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio StopEncoder
```

`PauseEncoder`

Description Pause audio encoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio PauseEncoder
```

`ResumeEncoder`

Description Resume audio encoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio ResumeEncoder
```

`StartDecoder`

Description Start audio decoder.

Execution

- Requires scheduler: Required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Audio decoder config. Example: { “type” :” PCM” ,” general” :{ “channels” :0,” sample_bits” :0,” sample_rate” :0,” frame_duration” :0}}

Schema JSON

CLI Command

```
svc_call Audio StartDecoder {"Config":null}
```

`StopDecoder`

Description Stop audio decoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio StopDecoder
```

`FeedDecoderData`

Description Feed audio data to the decoder.

Execution

- Requires scheduler: Not required

Parameters

- `Data`
  - **Type**: `RawBuffer`
  - **Required**: required
  - **Description**: Audio data to decode.

Schema JSON

CLI Command

```
svc_call Audio FeedDecoderData {"Data":null}
```

ResetData

Description Reset audio data. Includes player volume.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Audio ResetData
```

Events

PlayStateChanged

Description Emitted when playback state changes.

Execution

- Requires scheduler: Required

Items

- `State`
  - **Type**: `String`
  - **Description**: Playback state. Allowed values: [Idle, Playing, Paused]

Schema JSON

CLI Command

```
svc_subscribe Audio PlayStateChanged
```

AFE_EventHappened

Description Emitted when an AFE event occurs.

Execution

- Requires scheduler: Required

Items

- `Event`
  - **Type**: `String`
  - **Description**: AFE event. Allowed values: [VAD_Start, VAD_End, WakeStart, WakeEnd]

Schema JSON

CLI Command

```
svc_subscribe Audio AFE_EventHappened
```

`EncoderDataReady`

Description Emitted when encoder data is ready.

Execution

- Requires scheduler: Not required

Items

- `Data`
  - **Type**: `RawBuffer`
  - **Description**: Encoded audio data.

Schema JSON

CLI Command

```
svc_subscribe Audio EncoderDataReady
```

`RecorderDataReady`

Description Emitted when recorder data is ready.

Execution

- Requires scheduler: Not required

Items

- `Data`
  - **Type**: `RawBuffer`
  - **Description**: Recorded raw audio data.

Schema JSON

CLI Command

```
svc_subscribe Audio RecorderDataReady
```

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

4.2.5 Video

- Component registry: espressif/brookesia_service_video
- Helper header: `#include "brookesia/service_helper/video.hpp"`
- Helper class: `esp_brookesia::service::helper::Video`

Overview

brookesia_service_video provides:

- **Encoding**: Capture to compressed or raw formats (resolution, FPS, format per stream).
- **Decoding**: H.264, MJPEG, and others to RGB/YUV for display or processing.

Features

Encoder Input comes from a **local video device** (default `/dev/video0`; Kconfig can set path prefix and count).

Each output stream can be one of (per-stream resolution and FPS):

Type	Description
H.264	Common network/storage codec
MJPEG	Frame-wise JPEG-like compression
RGB565 / RGB888 / BGR888	RGB formats
YUV420 / YUV422 / O_UYY_E_VYY	YUV formats

> **Warning:** Multiple simultaneous outputs are not supported yet.

Decoder Compressed input and pixel output; configure per use case.

Input (compressed)

Format	Description
H.264	Common video codec
MJPEG	Frame-wise JPEG stream

Output (pixel)

Format	Description
RGB565 (BE/LE)	16-bit RGB for some panels/framebuffers
RGB888 / BGR888	24-bit RGB/BGR
YUV420P / YUV422P	Planar YUV for pipelines
YUV422 / UYVY422	Packed YUV for capture/display
O_UYY_E_VYY	Packed layout (hardware/pipeline dependent)

Standard Include / Helper Class

- Standard include: `#include \"brookesia/service_helper/video.hpp\"`
- Helper class: `esp_brookesia::service::helper::Video`

Service Interfaces

Functions

Encoder

`Open`

Description Open the encoder with config.

Execution

- Requires scheduler: Required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Encoder config. Example: { “sinks” :[{ “format” :” H264” ,” width” :320,” height” :240,” fps” :30},{ “format” :” MJPEG” ,” width” :320,” height” :240,” fps” :15}],” enable_stream_mode” :true}

Schema JSON

CLI Command

```
svc_call VideoEncoder0 Open {"Config":null}
```

`Close`

Description Close encoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call VideoEncoder0 Close
```

Start

Description Start encoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call VideoEncoder0 Start
```

Stop

Description Stop encoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call VideoEncoder0 Stop
```

FetchFrame

Description Fetch an encoder output frame and emit *FetchSinkFrameReady*. Only available in non-stream mode.

Execution

- Requires scheduler: Required

Parameters

- `SinkIndex`
  - **Type**: `Number`
  - **Required**: optional
  - **Default**: `0E0`
  - **Description**: Sink index.

Schema JSON

CLI Command

```
svc_call VideoEncoder0 FetchFrame {"SinkIndex":null}
```

Decoder

Open

Description Open the decoder with config.

Execution

- Requires scheduler: Required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Decoder config. Example: { “width” :320,” height” :240,” source_format” :” H264” ,” sink_format” :” RGB565_LE” ,” enable_stream_mode” :true,” enable_hw_acceleration” :true}

Schema JSON

CLI Command

```
svc_call VideoDecoder0 Open {"Config":null}
```

Close

Description Close decoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call VideoDecoder0 Close
```

`Start`

Description Start decoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call VideoDecoder0 Start
```

`Stop`

Description Stop decoder.

Execution

- Requires scheduler: Required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call VideoDecoder0 Stop
```

`FeedFrame`

Description Feed a decoder input frame.

Execution

- Requires scheduler: Required

Parameters

- `Frame`
  - **Type**: `RawBuffer`
  - **Required**: required
  - **Description**: Frame data.

Schema JSON

CLI Command

```
svc_call VideoDecoder0 FeedFrame {"Frame":null}
```

Events

Encoder

`StreamSinkFrameReady`

Description Emitted when an encoder stream frame is ready. Stream mode only.

Execution

- Requires scheduler: Not required

Items

- `SinkIndex`
  - **Type**: `Number`
  - **Description**: Sink index.
- `SinkInfo`
  - **Type**: `Object`
  - **Description**: Sink info. Example: { “format” :” H264” ,” width” :320,” height” :240,” fps” :30}
- `Frame`
  - **Type**: `RawBuffer`
  - **Description**: Encoded frame data.

Schema JSON

CLI Command

```
svc_subscribe VideoEncoder0 StreamSinkFrameReady
```

`FetchSinkFrameReady`

Description Emitted when an encoder fetched frame is ready. Non-stream mode only.

Execution

- Requires scheduler: Not required

Items

- `SinkIndex`
  - **Type**: `Number`
  - **Description**: Sink index.
- `SinkInfo`
  - **Type**: `Object`
  - **Description**: Sink info. Example: { “format” :” MJPEG” ,” width” :320,” height” :240,” fps” :15}
- `Frame`
  - **Type**: `RawBuffer`
  - **Description**: Encoded frame data.

Schema JSON

CLI Command

```
svc_subscribe VideoEncoder0 FetchSinkFrameReady
```

Decoder

`SinkFrameReady`

Description Emitted when a decoder output frame is ready.

Execution

- Requires scheduler: Not required

Items

- `Width`
  - **Type**: `Number`
  - **Description**: Decoded frame width.
- `Height`
  - **Type**: `Number`
  - **Description**: Decoded frame height.
- `Frame`
  - **Type**: `RawBuffer`
  - **Description**: Decoded frame data.

Schema JSON

CLI Command

```
svc_subscribe VideoDecoder0 SinkFrameReady
```

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

4.2.6 Custom Service

- Component registry: espressif/brookesia_service_custom
- Public header: `#include "brookesia/service_custom.hpp"`

Overview

brookesia_service_custom ships a ready-to-use **CustomService** so you can register custom functions and events without a separate Brookesia component.

Typical uses

- **Lightweight features**: LEDs, PWM, GPIO toggles, simple sensors—wrapped as CustomService calls.
- **Prototypes**: Expose logic as functions or events for local or RPC access.
- **Extensibility**: Add capabilities without changing the core service framework.

Features

- **Dynamic registration**: *register_function()* / *register_event()* at runtime.
- **Fixed handler shape**: *FunctionParameterMap* in, *FunctionResult* out; lambdas, *std::function*, free functions, functors, *std::bind*.
- **Events**: Full publish/subscribe lifecycle.
- **ServiceManager**: Local calls and remote RPC.
- **Optional worker**: Task scheduler for thread-safe execution.

API reference

Header File

- service/brookesia_service_custom/include/brookesia/service_custom/service_custom.hpp

Classes

class **`CustomService`** : public esp_brookesia::service::*ServiceBase*

Dynamic service that lets callers register functions and events at runtime.

Public Functions

bool **`register_function`**(FunctionSchema schema, FunctionHandler handler)

Register a function for custom service.

Parameters
- **`schema`** –**[in]** Function schema
- **`handler`** –**[in]** Function handler

Returns true if registered successfully, false otherwise

bool **`unregister_function`**(const std::string &function_name)

Unregister a function.

Parameters **`function_name`** –**[in]** Function name
Returns true if unregistered successfully, false otherwise

virtual std::vector<FunctionSchema> **get_function_schemas**() override

Get currently registered function schemas for custom service.

Returns std::vector<FunctionSchema> Function schemas

bool **register_event**(EventSchema event_schema)

Register an event for custom service.

Parameters **event_schema** –**[in]** Event schema to register.
Returns true if registered successfully, false otherwise

bool **unregister_event**(const std::string &event_name)

Unregister an event.

Parameters **event_name** –**[in]** Event name
Returns true if unregistered successfully, false otherwise

bool **publish_event**(const std::string &event_name, EventItemMap event_items)

Publish an event for custom service.

Parameters
- **event_name** –**[in]** Event name
- **event_items** –**[in]** Event items

Returns true if published successfully, false otherwise

virtual std::vector<EventSchema> **get_event_schemas**() override

Get currently registered event schemas for custom service.

Returns std::vector<EventSchema> Event schemas

Public Static Functions

static inline *CustomService* &**get_instance**()

Get the process-wide singleton instance.

Returns Reference to the singleton custom service.

4.3 Development guide

4.3.1 Usage

This guide explains how to integrate the ESP-Brookesia **service framework** in an ESP-IDF project, using the **Wi-Fi service** as an example. A full, buildable sample is in `examples/service/wifi`.

In a typical project, using a component directly means adding a dependency, including headers, and calling its APIs. As features grow, coupling between modules and components increases; removing a component then requires cleaning up headers and calls as well as the dependency, which raises maintenance cost.

ESP-Brookesia abstracts this behind two unified surfaces: **functions** and **events**. In many cases you only need `brookesia_service_manager` and `brookesia_service_helper` to reach service features through the Helper; you can also check at runtime whether a service is available. If the concrete service component is not linked, Helper usage usually still compiles, which helps with trimming and maintenance.

The framework also relies on mechanisms such as the *Task Scheduler* for asynchronous execution on the service side, reducing blocking on the caller. Exposed APIs are designed with thread safety in mind so application code does less ad-hoc synchronization.

The sections below follow this order: **dependencies → initialization and binding → calling functions → subscribing to events → event monitors**.

Adding component dependencies

Declare dependencies in `idf_component.yml` at the project root or in a component directory. For the Wi-Fi service:

```
dependencies:
  espressif/brookesia_service_wifi: "*"
  # espressif/brookesia_service_nvs: "*" # optional, if you need the NVS service
```

If your project does not link a concrete service implementation and you only need to satisfy "Helper code compiles", you can depend on `brookesia_service_helper` alone:

```
dependencies:
  espressif/brookesia_service_helper: "*"
```

For how to obtain components and version constraints, see *Obtaining and using components*.

Headers, namespace, and type aliases

```
#include "brookesia/lib_utils.hpp"
#include "brookesia/service_manager.hpp"
// Helper header: pick the Helper that matches your service component
#include "brookesia/service_helper/wifi.hpp"

// Brookesia data types live under the esp_brookesia namespace
using namespace esp_brookesia;
// Type aliases keep examples short
// Service-related types are under the service namespace
using WifiHelper = service::helper::Wifi;
```

Starting the service manager

Before calling any service API, initialize the global **singleton** `ServiceManager` by calling `start()`.

```
auto &service_manager = service::ServiceManager::get_instance();
service_manager.start();
```

Starting / stopping a service

```
// Before binding, you can check whether the Helper's service is linked into the␣
↪build
if (!WifiHelper::is_available()) {
    // Service unavailable: omitting the service component does not break␣
↪compilation, but this check fails
    return;
}

// Bind to the service: starts the service and its dependencies while `binding` is␣
↪alive;
// when `binding` goes out of scope it is destroyed and the service stops.
// To keep the service running longer, store `binding` outside the stack (e.g.␣
↪static or heap).
auto binding = service_manager.bind(WifiHelper::get_name().data());
if (!binding.is_valid()) {
    // Failed to start the service
    return;
}
```

Calling service functions

Before calling a function, confirm **parameter types, order, and return value** from the Helper contract or headers, for example:

- *Wi-Fi service function interface reference*
- Wi-Fi Helper header source

Note:

- **Description**: Function behavior, return information, etc. If there is no **return value** section, the function has no return value.
- **Parameter list: Parameter names, types, descriptions, and defaults.**
  - **Types: Six kinds are supported: `String`, `Number`, `Boolean`, `Object`, `Array`, `RawBuffer`. Mapping to c**
    - `String`: `std::string`
    - `Number`: `double` / `int` / *any arithmetic type*
    - `Boolean`: `bool`
    - `Object`: `boost::json::object` (often produced by serializing a `struct`)
    - `Array`: `boost::json::array` (often produced by serializing `std::vector`)
    - `RawBuffer`: `service::RawBuffer`
  - **Description:**
    - For `String` with enumerated values in the description, you pass enums via serialization.
    - For `Object` / `Array`, the description may include examples.
  - **Default**: Shown when the parameter is **optional**.
- **Execution requirement: Whether the function requires the Task Scheduler.**
  - **Required**: Must run **after the service has started**; **sync/async** calls go through the scheduler.
  - **Not required: May run before the service starts; only sync calls are allowed. Typical cases:**
    - The function must run before startup.
    - The parameter list contains `RawBuffer`.
    - The implementation is already thread-safe (locks, etc.), so the scheduler is not required.

Use static Helper methods for **synchronous** or **asynchronous** calls. **Synchronous** calls are described first.

Synchronous calls The call blocks until the function finishes or times out.

```
auto result = WifiHelper::call_function_sync<ReturnType>
↪(WifiHelper::FunctionId::FunctionName [, parameters...] [, timeout]);
if (!result) {
   // Call failed, log error
   BROOKESIA_LOGE("Failed: %1%", result.error());
} else {
   // Call succeeded
   // If the function returns a value, use `result.value()`
   auto &value = result.value(); // `value` has type `ReturnType`
   // ...
}
```

Note:

- `parameters` must match the schema in type, count, and order; omit them if the function has no parameters.
- If the return type is `void`, omit the `<ReturnType>` template argument.
- `timeout` is optional; if omitted, the default is `BROOKESIA_SERVICE_MANAGER_DEFAULT_CALL_FUNCTION_TIMEOU`
- See *`esp_brookesia::service::helper::Base::call_function_sync()`*.

Example: multiple parameters `SetConnectAp` —interface summary and sample code:

- Function: `SetConnectAp`
- **Parameters:**
  - `SSID`: `String`
  - `Password`: `String`
- Return type: `void`

```
// Pass parameters in schema order; do not skip required parameters
auto set_connect_ap_result = WifiHelper::call_function_sync(
   WifiHelper::FunctionId::SetConnectAp, "ssid1", "password1",␣
↪service::helper::Timeout(100));
if (!set_connect_ap_result) {
   // Call failed, log error
   BROOKESIA_LOGE("Failed: %1%", set_connect_ap_result.error());
}
```

Example: serialized parameters `TriggerGeneralAction` —interface summary and sample code:

- Function: `TriggerGeneralAction`
- **Parameters:**
  - `Action`: `String` (serialize from `WifiHelper::GeneralAction`)
- Return type: `void`

```
auto start_result = WifiHelper::call_function_sync(
   WifiHelper::FunctionId::TriggerGeneralAction,
   BROOKESIA_DESCRIBE_TO_STR(WifiHelper::GeneralAction::Start));
if (!start_result) {
   // Call failed, log error
   BROOKESIA_LOGE("Failed: %1%", start_result.error());
}
```

`SetScanParams` —interface summary and sample code:

- Function: `SetScanParams`
- **Parameters:**
  - `Param`: `Object` (serialize from `WifiHelper::ScanParams`)
- Return type: `void`

```
WifiHelper::ScanParams scan_params{
   .ap_count = 10,
   .interval_ms = 10000,
   .timeout_ms = 60000,
};
auto set_scan_params_result = WifiHelper::call_function_sync(
   WifiHelper::FunctionId::SetScanParams, BROOKESIA_DESCRIBE_TO_JSON(scan_params).
↪as_object());
if (!set_scan_params_result) {
   // Call failed, log error
   BROOKESIA_LOGE("Failed: %1%", set_scan_params_result.error());
}
```

Example: parsing return values `GetConnectedAps` —interface summary and sample code:

- Function: `GetConnectedAps`
- Parameters: none
- Return type: `boost::json::array` (deserialize to `std::vector<WifiHelper::ConnectApInfo>`)

```
auto get_connected_aps_result =
   WifiHelper::call_function_sync<boost::json::array>
↪(WifiHelper::FunctionId::GetConnectedAps);
if (!get_connected_aps_result) {
```

(continues on next page)

(continued from previous page)

```
    // Call failed, log error
    BROOKESIA_LOGE("Failed to get connected APs: %1%", get_connected_aps_result.
↪error());
}

// Parse return value
std::vector<WifiHelper::ConnectApInfo> infos;
auto parse_result = BROOKESIA_DESCRIBE_FROM_JSON(get_connected_aps_result.value(),␣
↪infos);
if (!parse_result) {
    // Parse failed, log error
    BROOKESIA_LOGE("Failed to parse connected APs: %1%", get_connected_aps_result.
↪error());
} else {
    // Parse succeeded, log result
    BROOKESIA_LOGI("Connected APs: %1%", infos);
}
```

Asynchronous calls The call **submits** work and returns immediately without blocking the caller. If you pass a result handler, the framework invokes it **asynchronously** when the function completes.

```
auto on_function_handler = [](service::FunctionResult &&result) {
    // Process return value
    if (!result.success) {
        // Execution failed, log error
        BROOKESIA_LOGE("Failed: %1%", result.error_message);
    } else {
        // Execution succeeded
        // If there is a return value:
        // auto &value = result.get_data<ReturnType>();
        // ...
    }
};
auto result = WifiHelper::call_function_async(WifiHelper::FunctionId::FunctionName␣
↪[, parameters...] [, on_function_handler]);
if (!result) {
    // Call failed
}
```

Note:

- `parameters` must match the schema; omit if the function has no parameters.
- `on_function_handler` is optional; if omitted, results are ignored.
- Serial async calls to the same service run **in submission order** inside the service:

```
Helper::call_function_async(Helper::FunctionId::Function1);
Helper::call_function_async(Helper::FunctionId::Function2);
Helper::call_function_async(Helper::FunctionId::Function3);
// Internal execution order: Function1 -> Function2 -> Function3
```

- See *esp_brookesia::service::helper::Base::call_function_async()*.

Example `GetConnectedAps` —interface summary and sample code:

- Function: `GetConnectedAps`
- Parameters: none
- Return type: `boost::json::array` (same as `std::vector<WifiHelper::ConnectApInfo>` when serialized)

```
auto on_get_connected_aps_handler = [](service::FunctionResult &&result) {
    if (!result.success) {
        // Execution failed, log error
        BROOKESIA_LOGE("Failed to get connected APs: %1%", result.error_message);
        return;
    }
    // Execution succeeded, parse return value
    std::vector<WifiHelper::ConnectApInfo> infos;
    auto &data = result.get_data<boost::json::array>();
    auto parse_result = BROOKESIA_DESCRIBE_FROM_JSON(data, infos);
    if (!parse_result) {
        // Parse failed, log error
        BROOKESIA_LOGE("Failed to parse connected APs: %1%", result);
        return;
    }
    // Parse succeeded, log result
    BROOKESIA_LOGI("Connected APs: %1%", infos);
};
auto get_connected_aps_result =
    WifiHelper::call_function_async(WifiHelper::FunctionId::GetConnectedAps, on_get_
↪connected_aps_handler);
if (!get_connected_aps_result) {
    // Call failed
    return;
}
```

Subscribing to service events

Confirm **item names, types, and order** from the Helper contract or headers, for example:

- *Wi-Fi service event interface reference*
- Wi-Fi Helper header source

Note:

- **Description**: Event behavior and return information. If there is no **return value** section, the event carries no return payload in that sense.
- **Item list: Item names, types, descriptions, and defaults.**
  - **Types: Same six kinds as for functions; mapping is the same:**
    - `String`: `std::string`
    - `Number`: `double` / `int` / *any arithmetic type*
    - `Boolean`: `bool`
    - `Object`: `boost::json::object` (often produced by serializing a `struct`)
    - `Array`: `boost::json::array` (often produced by serializing `std::vector`)
    - `RawBuffer`: `service::RawBuffer`
  - **Description:**
    - `String` with enumerated values deserializes to enum-like data.
    - `Object` / `Array` may include examples.
- **Execution requirement: Whether the event requires the Task Scheduler.**
  - **Required**: Must be published **after** the service has started.
  - **Not required: May be published before startup. Typical cases:**
    - The event must fire before startup.
    - The item list contains `RawBuffer`.

Subscribing to events Use `subscribe_event()` to register a callback; when the event fires, callbacks run **serially** in subscription order.

```
auto on_event_handler = [](const std::string &event_name[, items...]) {
    // Handle the event
};
// `conn` is the subscription; destruction **unsubscribes** (RAII).
// To keep the subscription alive, store `conn` outside the stack (e.g. static or␣
↪heap).
auto conn = WifiHelper::subscribe_event(WifiHelper::EventId::EventName, on_event_
↪handler);
if (!conn.connected()) {
    // Subscribe failed
}
// You can also unsubscribe manually
conn.disconnect();
```

Note:

- Call `subscribe_event()` only **after** `ServiceManager::start()`.
- `items` must match the event schema; omit if the event has no items.
- There is no hard limit on subscriptions; all callbacks for an event run in subscription order.
- See *esp_brookesia::service::helper::Base::subscribe_event()*.

Example `GeneralEventHappened` —interface summary and sample code:

- Event: `GeneralEventHappened`
- **Items:**
  - `Event`: `String` (deserialize to `WifiHelper::GeneralEvent`)
  - `IsUnexpected`: `Boolean`

```
auto on_general_event_happened_handler = [](const std::string &event_name, const␣
↪std::string &event,
                                            bool is_unexpected) {
    // Event fired, parse items
    WifiHelper::GeneralEvent general_event;
    auto parse_result = BROOKESIA_DESCRIBE_STR_TO_ENUM(event, general_event);
    if (!parse_result) {
        // Parse failed, log error
        BROOKESIA_LOGE("Failed to parse general event: %1%", event);
        return;
    }
    // Parse succeeded, log result
    BROOKESIA_LOGI("General event: %1%", general_event);
};
// Unsubscribe automatically when the connection is destroyed (RAII)
auto general_event_happened_connection = WifiHelper::subscribe_event(
    WifiHelper::EventId::GeneralEventHappened, on_general_event_happened_handler);
if (!general_event_happened_connection.connected()) {
    // Subscribe failed
    return;
}

// Trigger Start action
WifiHelper::call_function_async(WifiHelper::FunctionId::TriggerGeneralAction,
                                BROOKESIA_DESCRIBE_TO_
↪STR(WifiHelper::GeneralAction::Start));

// Wait for the event (example uses sleep; use a proper sync primitive in␣
↪production)
boost::this_thread::sleep_for(boost::chrono::seconds(5));
```

`ScanApInfosUpdated` —interface summary and sample code:

- Event: `ScanApInfosUpdated`
- **Items:**
  - `ApInfos`: `Array` (deserialize to `std::vector<WifiHelper::ScanApInfo>`)

```
auto on_scan_ap_infos_updated_handler = [](const std::string &event_name, const␣
↪boost::json::array &ap_infos) {
    // Event fired, parse items
    std::vector<WifiHelper::ScanApInfo> scanned_aps;
    auto parse_result = BROOKESIA_DESCRIBE_FROM_JSON(ap_infos, scanned_aps);
    if (!parse_result) {
        // Parse failed, log error
        BROOKESIA_LOGE("Failed to parse scan AP infos: %1%", ap_infos);
        return;
    }
    // Parse succeeded, log result
    BROOKESIA_LOGI("Scanned APs: %1%", scanned_aps);
};
auto scan_ap_infos_updated_connection =
    WifiHelper::subscribe_event(WifiHelper::EventId::ScanApInfosUpdated, on_scan_ap_
↪infos_updated_handler);
if (!scan_ap_infos_updated_connection.connected()) {
    // Subscribe failed
    return;
}

WifiHelper::call_function_async(WifiHelper::FunctionId::TriggerScanStart);

boost::this_thread::sleep_for(boost::chrono::seconds(10));
```

Event monitor `EventMonitor` adds **blocking wait** on top of **asynchronous events**: after triggering an API, you can wait for a matching event or any occurrence to drive control flow.

```
// Create and start the monitor
WifiHelper::EventMonitor<WifiHelper::EventId::EventName> event_monitor;
event_monitor.start();

// Call the API that raises this event (omitted)

// Wait for any occurrence
auto got_any_event = event_monitor.wait_for_any(timeout_ms);
if (!got_any_event) {
    // Timeout, event did not fire
    return;
}

// Or wait for an event whose items match
auto got_event = event_monitor.wait_for(std::vector<service::EventItem>{items...},␣
↪timeout_ms);
if (!got_event) {
    // Timeout, items did not match
    return;
}

event_monitor.stop();
```

Note:

- `items` must match the event definition in type, count, and order.
- See `esp_brookesia::service::helper::EventMonitor`.

Example: wait for specific event items `GeneralEventHappened` —monitor usage:

- Event: `GeneralEventHappened`
- **Items:**
  - `Event`: `String` (serialize from `WifiHelper::GeneralEvent`)
  - `IsUnexpected`: `Boolean`

```
WifiHelper::EventMonitor<WifiHelper::EventId::GeneralEventHappened> general_event_
↪monitor;
BROOKESIA_CHECK_FALSE_EXIT(general_event_monitor.start(), "Failed to start general␣
↪event monitor");

WifiHelper::call_function_async(WifiHelper::FunctionId::TriggerGeneralAction,
                                BROOKESIA_DESCRIBE_TO_
↪STR(WifiHelper::GeneralAction::Start));

auto got_event = general_event_monitor.wait_for(
    std::vector<service::EventItem>{BROOKESIA_DESCRIBE_TO_
↪STR(WifiHelper::GeneralEvent::Started), false}, 5000);
if (!got_event) {
    // Timeout, event did not fire
    return;
}
```

Example: latest received event items `ScanApInfosUpdated` —monitor usage:

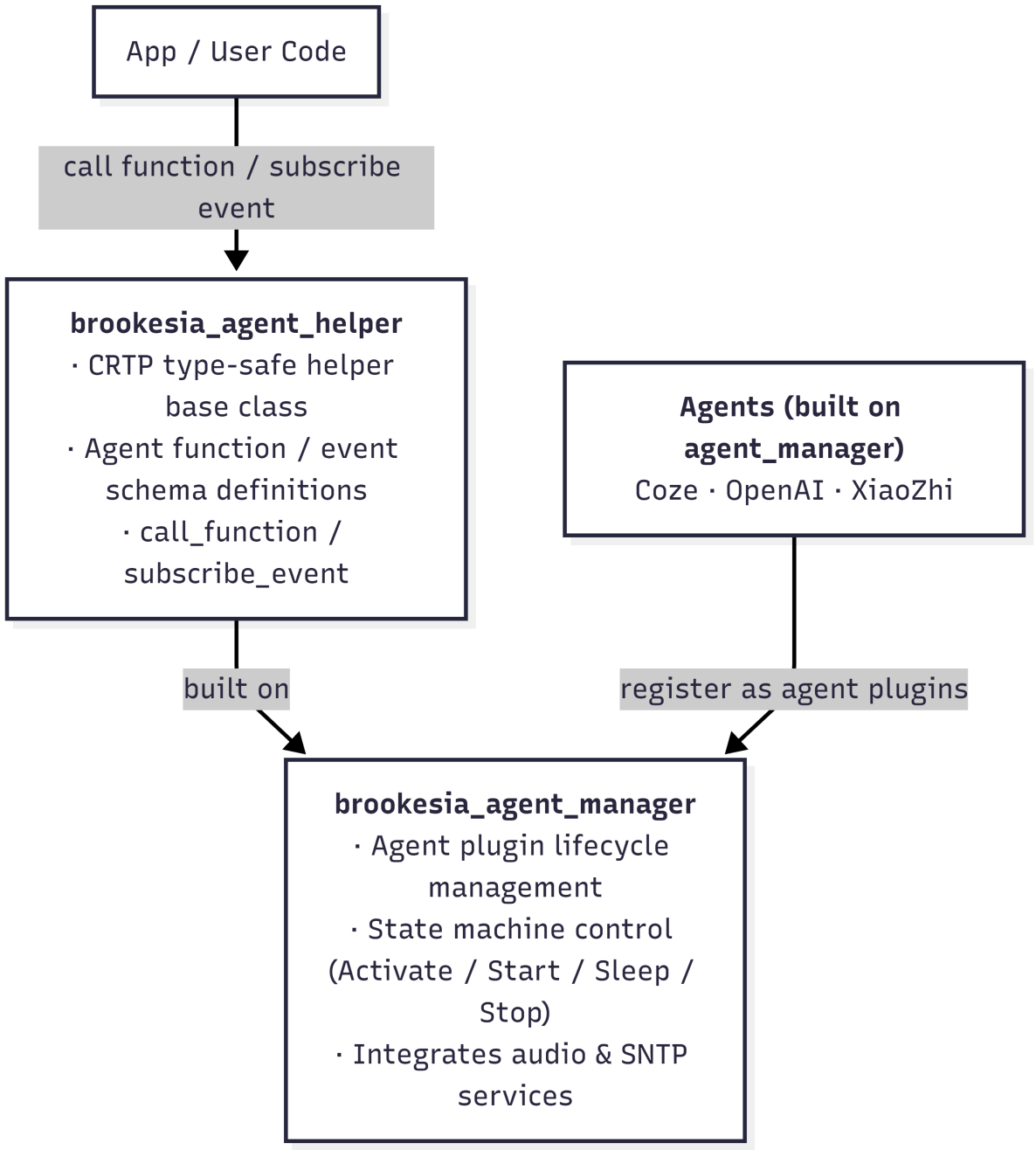
- Event: `ScanApInfosUpdated`
- **Items:**
  - `ApInfos`: `Array` (deserialize to `std::vector<WifiHelper::ScanApInfo>`)

```
WifiHelper::EventMonitor<WifiHelper::EventId::ScanApInfosUpdated> scan_ap_infos_
↪updated_monitor;
BROOKESIA_CHECK_FALSE_EXIT(scan_ap_infos_updated_monitor.start(), "Failed to start␣
↪scan AP infos updated monitor");

WifiHelper::call_function_async(WifiHelper::FunctionId::TriggerScanStart);

auto got_event = scan_ap_infos_updated_monitor.wait_for_any(10000);
if (!got_event) {
    // Timeout, event did not fire
    return;
}

auto last_items = scan_ap_infos_updated_monitor.get_last<boost::json::array>();
if (!last_items.has_value()) {
    return;
}
const auto &ap_infos = std::get<0>(last_items.value());
std::vector<WifiHelper::ScanApInfo> scanned_aps;
auto parse_result = BROOKESIA_DESCRIBE_FROM_JSON(ap_infos, scanned_aps);
if (!parse_result) {
    BROOKESIA_LOGE("Failed to parse scan AP infos: %1%", ap_infos);
    return;
}
BROOKESIA_LOGI("Scanned APs: %1%", scanned_aps);
```

Chapter 5

Agent Components

This section documents ESP-Brookesia agent framework components. The framework consists of an agent framework layer and a concrete-agents layer. Their component hierarchy is shown below:



- `brookesia_agent_manager`: The agent framework core, responsible for plugin registration, state machine lifecycle control, and integration of audio and time-sync services.
- `brookesia_agent_helper`: A CRTP-based type-safe helper layer that simplifies agent function/event definition and invocation, built on top of `service_helper`.
- **Agents**: Concrete AI platform integrations implemented on top of `agent_manager`, registered into the framework and managed uniformly by upper layers.

5.1 Agent framework

5.1.1 Agent Manager

- Component registry: espressif/brookesia_agent_manager

- Helper header: `#include "brookesia/agent_helper/manager.hpp"`
- Helper class: `esp_brookesia::agent::helper::Manager`

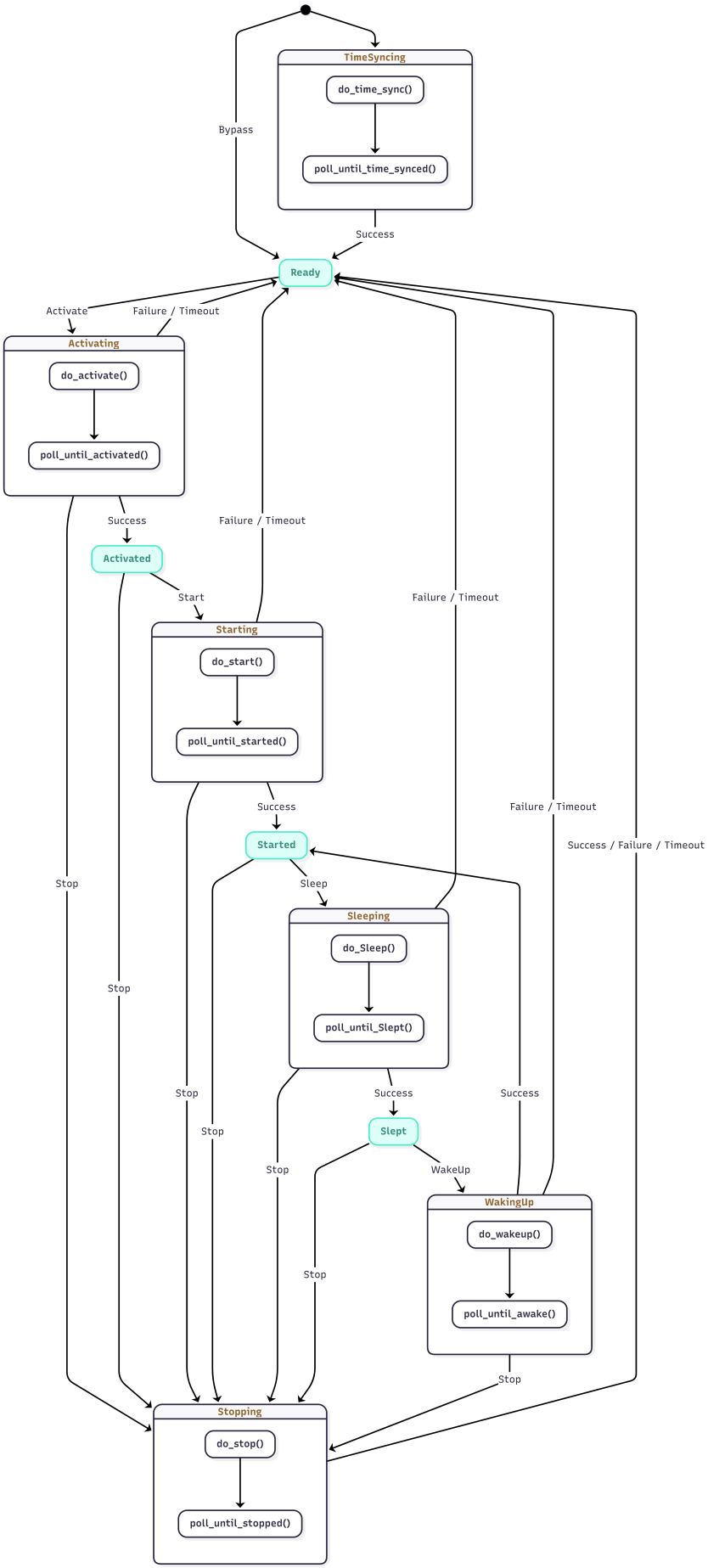
Overview

brookesia_agent_manager is the core agent framework:

- **Lifecycle**: Plugin-based init, activate, start, stop, sleep, wake, and dynamic switching.
- **State machine**: Correct, consistent transitions.
- **Control**: Pause/resume, interrupt, status queries.
- **Dialog modes**: RealTime and Manual (manual listen start/stop).
- **Events**: Operations, state, speech/listen, text, emotes—decoupled from apps.
- **Extensions**: Function calling, text, interrupt, emotes via agent properties.
- **AFE (optional)**: Wake begin/end handling.
- **Services**: Audio and SNTP integration.
- **Persistence**: Optional *brookesia_service_nvs* for active agent and mode.

State machine

The manager uses a state machine for agent lifecycle.



State diagram

States

State	Type	Description
TimeSyncing	Transient	Waiting for time sync.
Ready	Stable	Time OK (or skipped); waiting for activate.
Activating	Transient	Activating agent.
Activated	Stable	Activated; waiting for start.
Starting	Transient	Starting agent.
Started	Stable	Running; audio in/out enabled.
Sleeping	Transient	Entering sleep.
Slept	Stable	Asleep; can wake or stop.
WakingUp	Transient	Waking from sleep.
Stopping	Transient	Stopping agent.

API reference

Agent base Public header: `#include "brookesia/agent_manager/base.hpp"`

Header File

- agent/brookesia_agent_manager/include/brookesia/agent_manager/base.hpp

Classes

class **`Base`** : public esp_brookesia::service::*ServiceBase*

Common base class for all agent implementations.

The class extends *`service::ServiceBase`* with agent lifecycle hooks, audio pipeline coordination, and shared state tracked by *`Manager`* and `StateMachine`.

Subclassed by esp_brookesia::agent::Coze, esp_brookesia::agent::Openai, esp_brookesia::agent::XiaoZhi

Public Functions

inline const AgentAttributes &**`get_attributes`**`()` const

Get immutable metadata describing the agent.

Returns const AgentAttributes& Agent attributes passed at construction time.

inline const AudioConfig &**`get_audio_config`**`()` const

Get the audio configuration used by the agent.

Returns const AudioConfig& Current encoder and decoder configuration.

Agent manager Public header: `#include "brookesia/agent_manager/manager.hpp"`

Header File

- agent/brookesia_agent_manager/include/brookesia/agent_manager/manager.hpp

Classes

class **`Manager`** : public esp_brookesia::service::*ServiceBase*

Service responsible for managing agent selection and shared agent state.

Public Types

enum class **DataType**

Persistent keys managed by the agent manager.

Values:

enumerator **TargetAgent**

enumerator **ChatMode**

enumerator **Max**

Public Static Functions

static inline *Manager* &**get_instance**()

Get the global agent-manager singleton.

Returns *Manager*& Singleton instance.

5.1.2 Agent Helper

- Component registry: espressif/brookesia_agent_helper
- Public header: `#include "brookesia/agent_helper.hpp"`

Overview

brookesia_agent_helper uses C++20 concepts and CRTP to provide type-safe schemas and a unified calling surface for agents.

Features

- **Type-safe definitions**: Strong enums and structs with compile-time checks.
- **Schemas**: Standard function and event metadata (names, parameters, descriptions).
- **Calls**: Type-safe sync/async calls with conversion and error handling.
- **Events**: Type-safe subscriptions and callbacks.

Modules

Manager Helper

- Public header: `#include "brookesia/agent_helper/manager.hpp"`

Overview Raw Doxygen API for the Manager helper: types, enums, methods, and macros.

API reference

Header File

- agent/brookesia_agent_helper/include/brookesia/agent_helper/manager.hpp

Classes

class **Manager** : public esp_brookesia::service::helper::*Base*<*Manager*>

Helper schema definitions for the agent-manager service.

Public Types

enum class **AgentGeneralFunction** : uint8_t

Optional per-agent functions surfaced by the manager.

Values:

enumerator **InterruptSpeaking**

enumerator **Max**

enum class **AgentGeneralEvent** : uint8_t

Optional per-agent events surfaced by the manager.

Values:

enumerator **SpeakingStatusChanged**

enumerator **ListeningStatusChanged**

enumerator **AgentSpeakingTextGot**

enumerator **UserSpeakingTextGot**

enumerator **EmoteGot**

enumerator **Max**

enum class **ChatMode** : uint8_t

Conversation mode used by the active agent.

Values:

enumerator **RealTime**

enumerator **Manual**

enumerator **Max**

enum class **GeneralAction** : uint8_t

High-level actions accepted by the agent manager.

Values:

enumerator **TimeSync**

enumerator **Activate**

enumerator **Start**

enumerator **Sleep**

enumerator **WakeUp**

enumerator **Stop**

enumerator **Max**

enum class **GeneralEvent** : uint8_t

High-level completion events emitted by managed agents.

Values:

enumerator **TimeSynced**

enumerator **Activated**

enumerator **Started**

enumerator **Slept**

enumerator **Awake**

enumerator **Stopped**

enumerator **Max**

enum class **GeneralState** : uint8_t

State-machine states used by the agent manager.

Values:

enumerator **TimeSyncing**

enumerator **Ready**

enumerator **Activating**

enumerator **Activated**

enumerator **Starting**

enumerator **Started**

enumerator **Sleeping**

enumerator **Slept**

enumerator **WakingUp**

enumerator **Stopping**

enumerator **Max**

Public Static Functions

static inline constexpr std::string_view **get_name**()

Name of the agent-manager service.

Returns std::string_view Stable service name.

struct **AgentAttributes**

Public metadata describing one agent implementation.

Public Functions

inline std::string **get_name**() const

Get the agent name stored in the attributes.

Returns std::string Agent name.

inline bool **is_general_functions_supported**(*AgentGeneralFunction* function) const

Check whether an optional general function is supported.

Parameters **function** –**[in]** Function to check.

Returns true if the function is listed in `support_general_functions`.

inline bool **is_general_events_supported**(*AgentGeneralEvent* event) const

Check whether an optional general event is supported.

Parameters **event** –**[in]** Event to check.

Returns true if the event is listed in `support_general_events`.

Public Members

std::string **name**

Agent name exposed to the manager.

AgentOperationTimeout **operation_timeout** = {}

Timeout configuration for lifecycle actions.

std::vector<*AgentGeneralFunction*> **support_general_functions** = {}

Optional functions supported by the agent.

std::vector<*AgentGeneralEvent*> **support_general_events** = {}

Optional events emitted by the agent.

bool **require_time_sync** = false

Whether activation requires SNTP synchronization first.

struct **AgentOperationTimeout**

Timeout budget for the major lifecycle operations of an agent.

Public Members

uint32_t **activate** = 1000

Timeout for activation.

uint32_t **start** = 1000

Timeout for startup.

uint32_t **sleep** = 1000

Timeout for sleep.

uint32_t **wake_up** = 1000

Timeout for wake-up.

uint32_t **stop** = 1000

Timeout for shutdown.

Coze Helper

- Public header: `#include "brookesia/agent_helper/coze.hpp"`

Overview Raw Doxygen API for the Coze helper: types, enums, methods, and macros.

API reference

Header File

- agent/brookesia_agent_helper/include/brookesia/agent_helper/coze.hpp

Classes

class **Coze** : public esp_brookesia::service::helper::*Base*<*Coze*>

Helper schema definitions for the *Coze* agent service.

Public Types

enum class **CozeEvent**

Coze-specific events surfaced by the agent.

Values:

enumerator **InsufficientCreditsBalance**

enumerator **Max**

Public Static Functions

static inline constexpr std::string_view **get_name**()

Name of the *Coze* agent service.

Returns std::string_view Stable service name.

static inline std::span<const service::FunctionSchema> **get_function_schemas**()

Get the function schemas exported by the *Coze* agent.

Returns std::span<const service::FunctionSchema> Static schema span.

static inline std::span<const service::EventSchema> **get_event_schemas**()

Get the event schemas exported by the *Coze* agent.

Returns std::span<const service::EventSchema> Static schema span.

struct **AuthInfo**

Credentials required to authenticate with the *Coze* backend.

Public Members

std::string **session_name**

Session name used by the *Coze* SDK.

std::string **device_id**

Unique device identifier.

std::string **custom_consumer**

Consumer identifier passed to the backend.

std::string **app_id**

Application id.

std::string **user_id**

End-user id.

std::string **public_key**

Public key used for authentication.

std::string **private_key**

Private key used for authentication.

struct **Info**

Persistent *Coze* agent configuration.

Public Members

AuthInfo **authorization**

Authentication material.

std::vector<*RobotInfo*> **robots**

Available robot definitions.

struct **RobotInfo**

Metadata for one available *Coze* robot.

Public Members

std::string **name**

Human-readable robot name.

std::string **bot_id**

Coze bot identifier.

std::string **voice_id**

Voice profile identifier.

std::string **description**

Optional robot description.

OpenAI Helper

- Public header: `#include "brookesia/agent_helper/openai.hpp"`

Overview Raw Doxygen API for the OpenAI helper: types, enums, methods, and macros.

API reference

Header File

- agent/brookesia_agent_helper/include/brookesia/agent_helper/openai.hpp

Classes

class **Openai** : public esp_brookesia::service::helper::*Base*<*Openai*>

Helper schema definitions for the OpenAI agent service.

Public Static Functions

static inline constexpr std::string_view **get_name**()

Name of the OpenAI agent service.

Returns std::string_view Stable service name.

static inline std::span<const service::FunctionSchema> **get_function_schemas**()

Get the function schemas exported by the OpenAI agent.

Returns std::span<const service::FunctionSchema> Static schema span.

static inline std::span<const service::EventSchema> **get_event_schemas**()

Get the event schemas exported by the OpenAI agent.

Returns std::span<const service::EventSchema> Static schema span.

struct **Info**

Persistent configuration for the OpenAI agent backend.

Public Members

std::string **model**

Model identifier used for requests.

std::string **api_key**

API key used for authentication.

XiaoZhi Helper

- Public header: `#include "brookesia/agent_helper/xiaozhi.hpp"`

Overview Raw Doxygen API for the XiaoZhi helper: types, enums, methods, and macros.

API reference

Header File

- agent/brookesia_agent_helper/include/brookesia/agent_helper/xiaozhi.hpp

Classes

class **XiaoZhi** : public esp_brookesia::service::helper::*Base*<*XiaoZhi*>

Helper schema definitions for the *XiaoZhi* agent service.

Public Static Functions

static inline constexpr std::string_view **get_name**()

Name of the *XiaoZhi* agent service.

Returns std::string_view Stable service name.

static inline std::span<const service::FunctionSchema> **get_function_schemas**()

Get the function schemas exported by the *XiaoZhi* agent.

Returns std::span<const service::FunctionSchema> Static schema span.

static inline std::span<const service::EventSchema> **get_event_schemas**()

Get the event schemas exported by the *XiaoZhi* agent.

Returns std::span<const service::EventSchema> Static schema span.

5.2 Agents

5.2.1 Coze

- Component registry: espressif/brookesia_agent_coze
- Helper header: `#include "brookesia/agent_helper/coze.hpp"`
- Helper class: `esp_brookesia::agent::helper::Coze`

Overview

brookesia_agent_coze implements the Coze agent on top of ESP-Brookesia Agent Manager.

Features

- **Coze API**: WebSocket to Coze for real-time voice and text.
- **OAuth2**: Token acquisition and refresh for Coze.
- **Multi-bot**: Configure multiple bots and switch the active one.
- **Audio**: Built-in codecs; default G711A at 16 kHz.
- **Emotes**: Receive and display Coze expression events.
- **Platform events**: e.g. balance warnings.
- **Persistence**: Optional *brookesia_service_nvs* for auth and bot metadata.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/agent_helper/coze.hpp\"`
- Helper class: `esp_brookesia::agent::helper::Coze`

Service Interfaces

Functions

`SetActiveRobotIndex`

Description Set active robot index.

Execution

- Requires scheduler: Not required

Parameters

- `Index`
  - **Type**: `Number`
  - **Required**: required
  - **Description**: Robot index to activate.

Schema JSON

CLI Command

```
svc_call AgentCoze SetActiveRobotIndex {"Index":null}
```

GetActiveRobotIndex

Description Get active robot index. Return type: number. Example: 0

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call AgentCoze GetActiveRobotIndex
```

GetRobotInfos

Description Get robot info list. Return type: JSON array<object>. Example: [{ “name” :” robot1” ,” bot_id” :” bot_id1” ,” voice_id” :” voice_id1” ,” description” :” description1” },{ “name” :” robot2” ,” bot_id” :” bot_id2” ,” voice_id” :” voice_id2” ,” description” :” description2” }]

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call AgentCoze GetRobotInfos
```

Events

CozeEventHappened

Description Emitted when a Coze event occurs.

Execution

- Requires scheduler: Required

Items

- `CozeEvent`
  - **Type**: `String`
  - **Description**: Coze event. Allowed values: [InsufficientCreditsBalance]

Schema JSON

CLI Command

```
svc_subscribe AgentCoze CozeEventHappened
```

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

5.2.2 OpenAI

- Component registry: espressif/brookesia_agent_openai
- Helper header: `#include "brookesia/agent_helper/openai.hpp"`
- Helper class: `esp_brookesia::agent::helper::Openai`

Overview

brookesia_agent_openai implements the OpenAI agent on ESP-Brookesia Agent Manager.

Features

- **OpenAI Realtime**: WebRTC data channels for real-time voice and text.
- **P2P**: *esp_peer* for signaling and data channels.
- **Audio**: OPUS by default at 16 kHz, 24 kbps.
- **Data channel**: Audio, text, function calls, session control, etc.
- **Low latency**: Real-time duplex audio.
- **Persistence**: Optional *brookesia_service_nvs* for configuration.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/agent_helper/openai.hpp\"`
- Helper class: `esp_brookesia::agent::helper::Openai`

Service Interfaces

Functions This contract does not expose standard function schemas.

Events This contract does not publish standard event schemas.

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

5.2.3 XiaoZhi

- Component registry: espressif/brookesia_agent_xiaozhi
- Helper header: `#include "brookesia/agent_helper/xiaozhi.hpp"`
- Helper class: `esp_brookesia::agent::helper::XiaoZhi`

Overview

brookesia_agent_xiaozhi integrates the XiaoZhi AI platform for ESP-Brookesia.

Features

- **Platform**: Built on *esp_xiaozhi* SDK.
- **Voice**: OPUS at 16 kHz, 24 kbps.
- **Events**: Speaking/listening, agent/user text, emotes, etc.
- **Manual listen**: Start/stop listening for Manual mode.
- **Barge-in**: Interrupt agent speech.
- **Lifecycle**: Unified management via *brookesia_agent_manager*.

Standard Include / Helper Class

- Standard include: `#include \"brookesia/agent_helper/xiaozhi.hpp\"`
- Helper class: `esp_brookesia::agent::helper::XiaoZhi`

Service Interfaces

Functions

`AddMCP_ToolsWithServiceFunction`

Description Add MCP tools from service functions. Return type: JSON array<string> of added tool names. Example: [“Service.Audio.SetVolume” ,” Service.Audio.GetVolume”]

Execution

- Requires scheduler: Not required

Parameters

- `ServiceName`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Service name.
- `FunctionNames`
  - **Type**: `Array`
  - **Required**: optional
  - **Default**: `[]`

- **Description**: Function names as JSON array<string>. Empty means all functions in the service. Example: [“SetVolume” ,” GetVolume”]

Schema JSON

CLI Command

```
svc_call AgentXiaoZhi AddMCP_ToolsWithServiceFunction {"ServiceName":null,
↪"FunctionNames":null}
```

AddMCP_ToolsWithCustomFunction

Description Add custom MCP tools. Return type: JSON array<string> of added tool names. Example: [“Custom.Display.GetBrightness” ,” Custom.Display.SetBrightness”]

Execution

- Requires scheduler: Not required

Parameters

- `Tools`
  - **Type**: `Array`
  - **Required**: required
  - **Description**: Tools to add as JSON array<object>. Example: [{ “description” :” custom tool description 1” ,” name” :” Display.GetBrightness” },{ “description” :” custom tool description 2” ,” name” :” Display.SetBrightness” }]

Schema JSON

CLI Command

```
svc_call AgentXiaoZhi AddMCP_ToolsWithCustomFunction {"Tools":null}
```

RemoveMCP_Tools

Description Remove MCP tools.

Execution

- Requires scheduler: Not required

Parameters

- `Tools`
  - **Type**: `Array`
  - **Required**: required
  - **Description**: Tool names to remove. Example: [“Service.Audio.SetVolume” ,” Custom.Display.GetBrightness”]

Schema JSON

CLI Command

```
svc_call AgentXiaoZhi RemoveMCP_Tools {"Tools":null}
```

ExplainImage

Description Explain an image. Return type: string. Example: "This image contains a cup on a desk."

Execution

- Requires scheduler: Required

Parameters

- `Image`
  - **Type**: `RawBuffer`
  - **Required**: required
  - **Description**: Image data.
- `Question`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `"What is in the image?"`
  - **Description**: Question text.

Schema JSON

CLI Command

```
svc_call AgentXiaoZhi ExplainImage {"Image":null,"Question":null}
```

Events

ActivationCodeReceived

Description Emitted when an activation code is received.

Execution

- Requires scheduler: Required

Items

- `Code`
  - **Type**: `String`
  - **Description**: Activation code.

Schema JSON

CLI Command

```
svc_subscribe AgentXiaoZhi ActivationCodeReceived
```

Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

Chapter 6

Expression Components

This section documents ESP-Brookesia AI expression components.

6.1 Emote

- Component registry: espressif/brookesia_expression_emote
- Helper header: `#include "brookesia/service_helper/expression/emote.hpp"`
- Helper class: `esp_brookesia::service::helper::ExpressionEmote`

6.1.1 Overview

brookesia_expression_emote manages emotes on the Brookesia service framework:

- **Resources**: Load and configure emoji/animation assets.
- **Emotes**: Set and manage emoji for emotional states.
- **Animation**: Playback control; wait for frames; stop.
- **QR codes**: Show and hide QR overlays.
- **Event messages**: Show and hide event text.

6.1.2 Standard Include / Helper Class

- Standard include: `#include \"brookesia/service_helper/expression/emote.hpp\"`
- Helper class: `esp_brookesia::service::helper::ExpressionEmote`

6.1.3 Service Interfaces

Functions

`SetConfig`

Description Set emote config.

Execution

- Requires scheduler: Not required

Parameters

- `Config`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Config. Example: {"h_res":320,"v_res":240,"buf_pixels":7680,"fps":30,"task_priority":5," task_stack" :4096," task_affinity" :0," task_stack_in_ext" :true," flag_swap_color_bytes" :false," flag_double_buffer" :false," flag_buff_dma" :false," flag_buff_spiram" :true}

Schema JSON

CLI Command

```
svc_call Emote SetConfig {"Config":null}
```

`LoadAssetsSource`

Description Load assets from the specified source.

Execution

- Requires scheduler: Not required

Parameters

- `Source`
  - **Type**: `Object`
  - **Required**: required
  - **Description**: Asset source as a JSON object. Example: { "source" :" anim_icon" ," type" :" PartitionLabel" ," flag_enable_mmap" :false}

Schema JSON

CLI Command

```
svc_call Emote LoadAssetsSource {"Source":null}
```

`SetEmoji`

Description Set emoji and hide animation immediately.

Execution

- Requires scheduler: Not required

Parameters

- `Emoji`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Emoji name.

Schema JSON

CLI Command

```
svc_call Emote SetEmoji {"Emoji":null}
```

HideEmoji

Description Hide current emoji.

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Emote HideEmoji
```

SetAnimation

Description Set animation and hide emoji immediately.

Execution

- Requires scheduler: Not required

Parameters

- `Animation`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Animation name.

Schema JSON

CLI Command

```
svc_call Emote SetAnimation {"Animation":null}
```

InsertAnimation

Description Insert animation; it hides immediately and shows after the duration.

Execution

- Requires scheduler: Not required

Parameters

- `Animation`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Animation name.
- `DurationMs`
  - **Type**: `Number`
  - **Required**: required
  - **Description**: Animation duration in milliseconds. Stops automatically after this duration.

Schema JSON

CLI Command

```
svc_call Emote InsertAnimation {"Animation":null,"DurationMs":null}
```

StopAnimation

Description Stop current animation and hide it immediately.

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Emote StopAnimation
```

WaitAnimationFrameDone

Description Wait for each animation frame to finish.

Execution

- Requires scheduler: Not required

Parameters

- `TimeoutMs`
  - **Type**: `Number`
  - **Required**: optional
  - **Default**: `0E0`
  - **Description**: Timeout in milliseconds. *0* means wait forever.

Schema JSON

CLI Command

```
svc_call Emote WaitAnimationFrameDone {"TimeoutMs":null}
```

SetEventMessage

Description Set message for a specified emote event.

Execution

- Requires scheduler: Not required

Parameters

- `Event`
  - **Type**: `String`
  - **Required**: required
  - **Description**: Event type. Allowed values: [Idle, Speak, Listen, System, User, Battery]
- `Message`
  - **Type**: `String`
  - **Required**: optional
  - **Default**: `""`
  - **Description**: Message text.

Schema JSON

CLI Command

```
svc_call Emote SetEventMessage {"Event":null,"Message":null}
```

HideEventMessage

Description Hide current event message.

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Emote HideEventMessage
```

SetQrcode

Description Set QR code and hide emoji and animation immediately.

Execution

- Requires scheduler: Not required

Parameters

- `Qrcode`
  - **Type**: `String`
  - **Required**: required
  - **Description**: QR code content.

Schema JSON

CLI Command

```
svc_call Emote SetQrcode {"Qrcode":null}
```

HideQrcode

Description Hide current QR code and show emoji immediately.

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Emote HideQrcode
```

NotifyFlushFinished

Description Notify emote flush finished.

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Emote NotifyFlushFinished
```

`RefreshAll`

Description Refresh the screen.

Execution

- Requires scheduler: Not required

Parameters

- No parameters.

Schema JSON

CLI Command

```
svc_call Emote RefreshAll
```

Events

`FlushReady`

Description Emitted when emote flush is ready.

Execution

- Requires scheduler: Not required

Items

- `Param`
  - **Type**: `Object`
  - **Description**: Flush-ready parameter as a JSON object. Example: { “x_start” :0,” y_start” :0,” x_end” :100,” y_end” :100,” data” :” @0x12345678” }

Schema JSON

CLI Command

```
svc_subscribe Emote FlushReady
```

6.1.4 Related Types and Configs

For structs, enums, helper methods, and the full Doxygen-rendered API surface, see *Raw helper API reference*.

Index

Symbols

B

E

esp_brookesia::agent::helper::Coze::CozeEvent (*C++ enum*), 198

esp_brookesia::agent::helper::Coze::CozeEvent::InsufficientCreditsBalance (*C++ enumerator*), 198

esp_brookesia::agent::helper::Coze::CozeEvent::Max (*C++ enumerator*), 198

esp_brookesia::agent::helper::Coze::get_event_schemas (*C++ function*), 199

esp_brookesia::agent::helper::Coze::get_function_schemas (*C++ function*), 199

esp_brookesia::agent::helper::Coze::get_name (*C++ function*), 199

esp_brookesia::agent::helper::Coze::Info (*C++ struct*), 199

esp_brookesia::agent::helper::Coze::Info::authorization (*C++ member*), 200

esp_brookesia::agent::helper::Coze::Info::robots (*C++ member*), 200

esp_brookesia::agent::helper::Coze::RobotInfo (*C++ struct*), 200

esp_brookesia::agent::helper::Coze::RobotInfo::bot_id (*C++ member*), 200

esp_brookesia::agent::helper::Coze::RobotInfo::description (*C++ member*), 200

esp_brookesia::agent::helper::Coze::RobotInfo::name (*C++ member*), 200

esp_brookesia::agent::helper::Coze::RobotInfo::voice_id (*C++ member*), 200

esp_brookesia::agent::helper::Manager (*C++ class*), 195

esp_brookesia::agent::helper::Manager::AgentAttributes (*C++ struct*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::get_name (*C++ function*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::is_general_events_supported (*C++ function*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::is_general_functions_supported (*C++ function*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::name (*C++ member*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::operation_timeout (*C++ member*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::require_time_sync (*C++ member*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::support_general_events (*C++ member*), 197

esp_brookesia::agent::helper::Manager::AgentAttributes::support_general_functions (*C++ member*), 197

esp_brookesia::agent::helper::Manager::AgentGeneralEvent (*C++ enum*), 195

esp_brookesia::agent::helper::Manager::AgentGeneralEvent::AgentSpeakingTextGot (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGeneralEvent::EmoteGot (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGeneralEvent::ListeningStatusChanged (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGeneralEvent::Max (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGener (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGener (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGener (*C++ enum*), 195

esp_brookesia::agent::helper::Manager::AgentGener (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentGener (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::AgentOpera (*C++ struct*), 198

esp_brookesia::agent::helper::Manager::AgentOpera (*C++ member*), 198

esp_brookesia::agent::helper::Manager::AgentOpera (*C++ member*), 198

esp_brookesia::agent::helper::Manager::AgentOpera (*C++ member*), 198

esp_brookesia::agent::helper::Manager::AgentOpera (*C++ member*), 198

esp_brookesia::agent::helper::Manager::AgentOpera (*C++ member*), 198

esp_brookesia::agent::helper::Manager::ChatMode (*C++ enum*), 195

esp_brookesia::agent::helper::Manager::ChatMode:: (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::ChatMode:: (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::ChatMode:: (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enum*), 195

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 195

esp_brookesia::agent::helper::Manager::GeneralAct (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralEve (*C++ enum*), 196

esp_brookesia::agent::helper::Manager::GeneralEve (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralEve (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralEve (*C++ enumerator*), 196

esp_brookesia::agent::helper::Manager::GeneralEve (*C++ enumerator*), 196