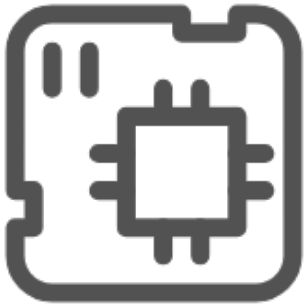
ESP-Brookesia 编程指南

Release master
乐鑫信息科技
2026 年 04 月 13 日

Table of contents

索引 **211**

ESP-BROOKESIA

		
快速开始	工具组件	硬件抽象组件
		
服务组件	AI 智能体组件	AI 表达组件

概述

ESP-Brookesia 是一个面向 AIoT 设备的人机交互开发框架，旨在简化应用程序开发和 AI 能力集成的流程。它以 ESP-IDF 为基础，通过组件化架构为开发者提供从硬件抽象、系统服务到 AI 智能体的全栈支持，加速 HMI 与 AI 应用产品的开发与上市。

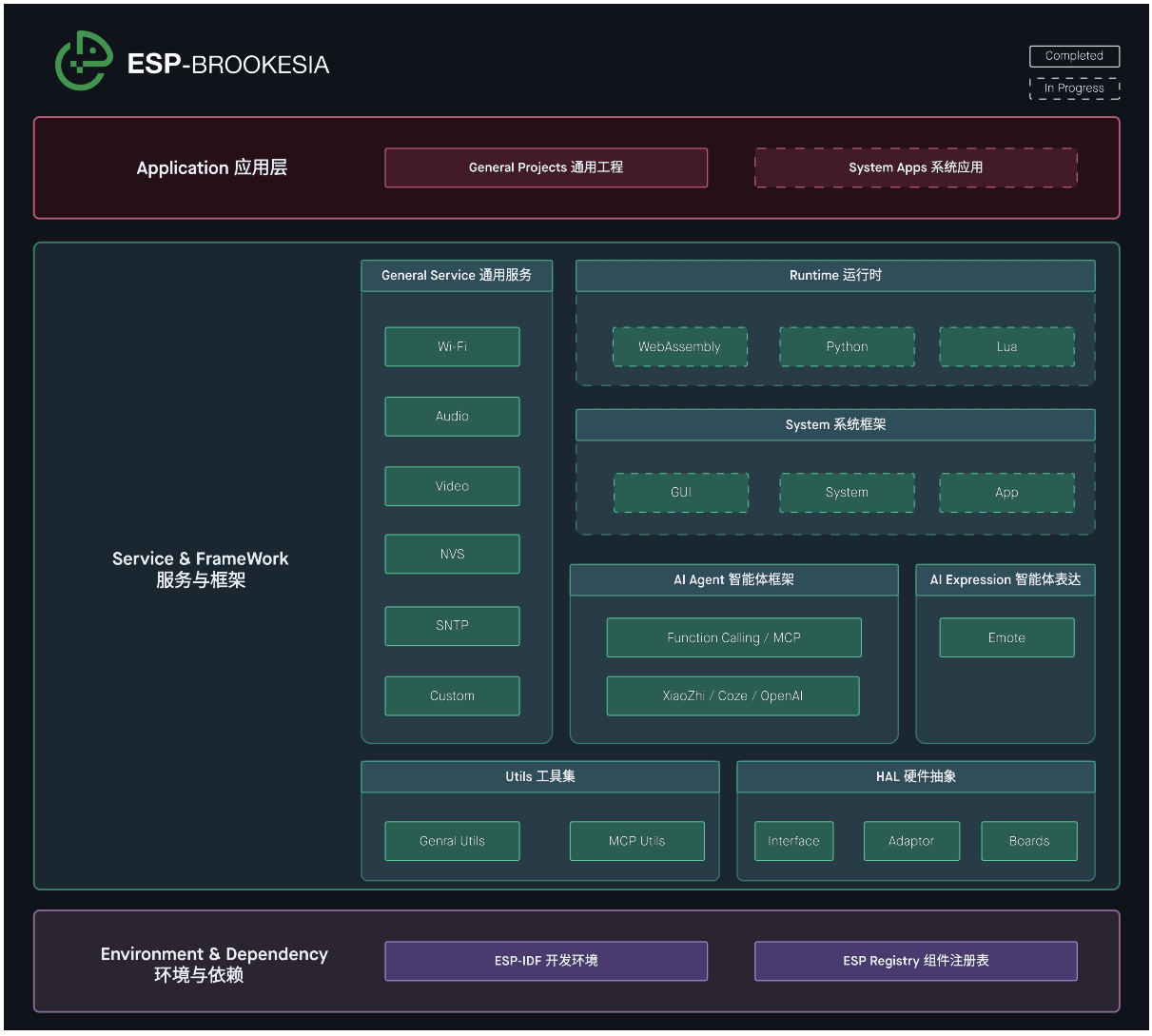
备注： “Brookesia”是一种变色龙属的物种，擅长于伪装和适应环境，这与 ESP-Brookesia 的目标紧密相关。该框架旨在提供一种灵活、可扩展的解决方案，能够适应各种不同的硬件设备和应用需求，就像 Brookesia 变色龙那样，具有高度的适应性和灵活性。

ESP-Brookesia 的主要特性包括：

- **原生 ESP-IDF 支持**：基于 C/C++ 开发，深度集成 ESP-IDF 开发体系与 ESP Registry 组件注册表，充分利用乐鑫开源组件生态。
- **可扩展的硬件抽象**：定义统一的硬件接口（音频、显示、触摸、存储等），并提供板级适配层，便于快速移植到不同硬件平台。
- **丰富的系统服务**：提供 Wi-Fi 连接、音视频处理等开箱即用的系统级服务，采用 Manager + Helper 架构实现解耦与扩展，为 Agent CLI 提供支持。
- **多 LLM 后端集成**：内置对 OpenAI、Coze、小智等主流 AI 平台的适配，提供统一的智能体管理与生命周期控制。
- **MCP 协议支持**：通过 Function Calling / MCP 协议将设备服务能力暴露给大语言模型，实现 LLM 与系统服务的统一通信。
- **AI 表达能力**：支持表情集、动画集等可视化 AI 表达，为拟人化交互提供丰富的视觉反馈。

功能架构

ESP-Brookesia 采用分层架构设计，自底向上由 **环境与依赖**、**服务与框架**以及 **应用层**三个层级组成，如下图所示：



环境与依赖

框架的运行基础。ESP-IDF 开发环境提供编译工具链、实时操作系统及外设驱动等底层支撑；ESP Registry 组件注册表统一管理框架各组件及其第三方依赖的分发与版本迭代。

服务与框架

框架的核心层，向下对接环境与依赖，向上为应用程序和 AI 智能体提供标准化的服务接口，涵盖基础工具、硬件抽象、系统服务、AI 智能体及表达等模块。

- **Utils 工具集**：为上层模块提供通用基础能力。其中 General Utils 包含日志系统、错误检查、状态机、任务调度器、插件管理、内存/线程/时间分析器等核心工具；MCP Utils 作为 ESP-Brookesia 服务体系与 MCP 引擎之间的桥梁，将已注册的服务函数暴露为标准 MCP 工具，实现大语言模型对设备能力的调用。
- **HAL 硬件抽象**：定义统一的硬件访问接口并提供板级适配实现。其中 Interface 定义音频播放/录制、显示面板/触摸、状态 LED、存储文件系统等标准化硬件接口；Adaptor 针对具体开发板提供接口实现，完成硬件资源的初始化与映射。Boards 提供板级 YAML 配置，描述各开发板的外设拓扑、引脚与驱动参数。
- **General Service 通用服务**：提供系统级基础服务，包括 Wi-Fi 连接管理、Audio 音频采集与播放、Video 视频编解码、NVS 非易失性存储、SNTP 网络时间同步，以及 Custom 自定义服务扩展机制。所有服务均基于 Manager + Helper 架构，支持本地调用与 RPC 远程通信。
- **AI Agent 智能体框架**：提供 AI 智能体的统一管理框架，内置对 Coze、OpenAI、小智等主流 AI 平台的适配。通过 Function Calling / MCP 协议实现大语言模型与系统服务的双向通信，使 LLM 能够感知和调用设备的各项能力。
- **AI Expression 智能体表达**：提供 AI 交互场景下的可视化表达能力，包括 Emote 表情集与动画控制，为拟人化交互提供丰富的视觉反馈。
- **System 系统框架** *(规划中)*：面向不同产品形态（移动设备、音箱、机器人等）提供 GUI、系统管理与应用框架支持。
- **Runtime 运行时** *(规划中)*：提供 WebAssembly、Python、Lua 等脚本运行时支持，实现应用的动态加载与执行。

应用层

基于上述各层构建的最终产品与工程：

- General Projects 通用工程：面向产品的通用工程模板，集成框架各组件，可直接用于产品开发。
- System Apps 系统应用 *(规划中)*：面向产品的系统级应用集合，包括设置、AI 助手、应用商店等，可按需裁剪和集成。

Chapter 1

快速入门

本文档介绍如何获取和使用 ESP-Brookesia 组件，以及如何编译和运行示例工程。

1.1 ESP-Brookesia 版本说明

ESP-Brookesia 自 `v0.7` 版本起采用组件化管理，建议项目通过组件注册表获取所需组件，约定如下：

1. 各组件独立迭代，但均具有相同的 `major.minor` 版本号，并且依赖相同的 ESP-IDF 版本。
2. `release` 分支仅维护历史大版本，`master` 分支持续集成新特性。

不同版本说明如下，组件列表与更新说明请参考：

表 1: ESP-Brookesia 版本支持情况

ESP-Brookesia	依赖的 ESP-IDF	主要变更	支持状态
master (v0.7)	>= v5.5, < 6.0	支持组件管理器	新功能开发分支
release/v0.6	>= v5.3, <= 5.5	预览支持系统框架，提供 ESP-VoCat 固件工程	停止维护

1.2 开发环境搭建

ESP-IDF 是乐鑫为 ESP 系列芯片提供的物联网开发框架：

- ESP-IDF 包含一系列库及头文件，提供了基于 ESP SoC 构建软件项目所需的核心组件；
- ESP-IDF 还提供了开发和量产过程中最常用的工具及功能，例如：构建、烧录、调试和测量等。

备注：

- 请参考：ESP-IDF 编程指南 完成 ESP-IDF 开发环境的搭建。
- 不推荐使用 VSCode 扩展插件安装 ESP-IDF 环境，可能导致部分依赖 *esp_board_manager* 组件的示例编译失败。

1.3 硬件准备

ESP 系列 SoC 支持以下功能：

- Wi-Fi（2.4 GHz/5 GHz 双频）
- 蓝牙 5.x（BLE/Mesh）

- 高性能多核处理器，主频最高可达 400 MHz
- 超低功耗协处理器和深度睡眠模式
- **丰富的外设接口：**
  - 通用接口：GPIO、UART、I2C、I2S、SPI、SDIO、USB OTG 等
  - 专用接口：LCD、摄像头、以太网、CAN、Touch、LED PWM、温度传感器等
- **大容量内存：**
  - 内部 RAM 最大可达 768 KB
  - 支持外部 PSRAM 扩展
  - 支持外部 Flash 存储
- **增强的安全特性：**
  - 硬件加密引擎
  - 安全启动
  - Flash 加密
  - 数字签名

ESP 系列 SoC 采用先进工艺制程，提供业界领先的射频性能、低功耗特性和稳定可靠性，适用于物联网、工业控制、智能家居、可穿戴设备等多种应用场景。

备注：

- 各系列芯片的具体规格和功能请参考 ESP 产品选型工具。
- 参考 ESP-Brookesia HAL 开发板支持 快速开始
- ESP-Brookesia 对内存和 Flash 占用较高，建议选用 Flash ≥ 8 MB、PSRAM ≥ 4 MB 的芯片或开发板。

1.4 如何获取和使用组件

推荐通过 ESP Component Registry 获取 ESP-Brookesia 组件。

以 `brookesia_service_wifi` 组件为例，添加依赖的步骤如下：

1. **使用命令行**
   在工程目录下运行以下命令：

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

2. **修改配置文件**
   在工程目录下创建或修改 *idf_component.yml* 文件：

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

更多组件管理器的使用方法请参考 ESP Registry Docs。

1.5 如何使用示例工程

ESP-Brookesia 提供了多个示例工程。示例工程的使用方法如下：

1. 确保已经完成 ESP-IDF 开发环境的搭建
2. 选择目标芯片或开发板。根据功能依赖的外设不同，该步骤可分为：
   - **选择目标芯片**：
     以 `examples/service/wifi` 示例工程为例，该工程仅依赖于芯片的 WiFi 外设功能，因此仅需选择目标芯片（如 `esp32s3` 等）：

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

   - **选择目标开发板**：
     以 `examples/service/console` 示例工程为例，该工程依赖于开发板的音频外设功能，因此需要选择目标开发板（如 `esp_vocat_board_v1_2` 等）：

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

3. 配置项目（可选）：

```
idf.py menuconfig
```

4. 编译并烧录到开发板：

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

5. 通过串口监视输出：

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

更多示例请见 `examples/` 目录，具体使用方法请参考各示例目录下的 README 文件。

Chapter 2

工具

本分类包含 ESP-Brookesia 工具类组件的说明内容。

2.1 通用工具类

- 组件注册表：espressif/brookesia_lib_utils
- 公共头文件：`#include "brookesia/lib_utils.hpp"`

2.1.1 概述

brookesia_lib_utils 是 ESP-Brookesia 框架的通用工具库，提供任务调度、线程配置、性能分析、日志、状态机、插件系统以及通用辅助工具等能力。

2.1.2 特性

- 线程配置：RAII 风格线程配置，支持名称、优先级、栈大小、核心绑定等
- 任务调度器：基于 Boost.Asio 的异步任务调度，支持立即、延迟与周期任务
- 日志系统：支持 ESP_LOG 与 printf 输出，并支持格式化
- 状态机与插件系统：支持复杂状态流转和模块化扩展
- 辅助工具：覆盖错误检查、函数守卫、对象描述与序列化等能力
- 性能分析工具：提供内存、线程、时间维度分析能力

2.1.3 核心工具

线程配置

公共头文件：`#include "brookesia/lib_utils/thread_config.hpp"`

概述 *thread_config* 提供线程运行参数的统一配置能力，并通过 RAII 守卫在作用域内自动应用与恢复线程配置。

特性

- 提供 *ThreadConfig* 结构体，统一管理名称、优先级、栈大小、核心绑定等
- 支持从系统默认配置或当前线程配置读取并应用
- *ThreadConfigGuard* 作用域内自动恢复原配置
- 提供便捷宏用于配置与查询线程参数

API 参考

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.

参数 **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
```

参数

- ... –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());
```

备注： 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.

返回 `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());
```

备注： 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.

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

任务调度器

公共头文件：`#include "brookesia/lib_utils/task_scheduler.hpp"`

概述 *task_scheduler* 提供基于 Boost.Asio 的任务调度器，支持即时、延迟、周期任务以及分组串行/并行执行。

特性

- 支持一次性、延迟、周期与批量任务调度
- 任务分组与串行执行控制
- 支持暂停、恢复、取消与等待任务完成
- 提供统计信息与执行前后回调

API 参考

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.

备注： 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.

备注： 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.

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

inline bool **start**()

Start the scheduler with the default configuration.

返回 `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.

返回 `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.

参数

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

返回 `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.

参数

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

返回 `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.

参数

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

返回 `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.

参数

- **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.

返回 `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.

参数

- **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.

返回 `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.

参数
- **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.

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

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

Cancel a task by ID.

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

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

Cancel every task in a group.

参数 **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.

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

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

Suspend all suspendable tasks in a group.

参数 **group** –**[in]** Group name.
返回 Number of tasks suspended.

size_t **suspend_all**()

Suspend all suspendable tasks.

返回 Number of tasks suspended.

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

Resume a suspended delayed or periodic task.

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

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

Resume all suspended tasks in a group.

参数 **group** –**[in]** Group name.
返回 Number of tasks resumed.

size_t **resume_all**()

Resume all suspended tasks.

返回 Number of tasks resumed.

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

Wait for a task to complete.

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

返回 `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.

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

返回 `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.

参数 **timeout_ms** –**[in]** Timeout in milliseconds. `-1` waits indefinitely.
返回 `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.

参数 **id** –**[in]** Task ID whose timer should restart.
返回 `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.

参数 **id** –**[in]** Task ID.
返回 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.

参数 **id** –**[in]** Task ID.
返回 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.

参数 **id** –**[in]** Task ID.
返回 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.

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

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

Get all groups that currently contain tasks.

返回 Vector of active group names.

Statistics **get_statistics**() const

Get execution statistics accumulated by the scheduler.

返回 *`Statistics`* structure with task counters.

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

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

返回 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.

返回 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.

状态基类

公共头文件：`#include "brookesia/lib_utils/state_base.hpp"`

概述 *state_base* 定义状态机的基础状态抽象，提供进入、退出、更新等生命周期回调，并支持状态超时与更新间隔配置。

特性

- 标准化状态生命周期回调：*on_enter* / *on_exit* / *on_update*
- 支持超时动作与更新间隔设置
- 便于派生自定义状态类，统一状态行为管理

API 参考

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.

参数 **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.

参数

- **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.

返回 `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.

参数

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

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

inline virtual void **on_update**()

Hook invoked periodically while the state remains active.

备注: 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.

参数

- **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.

参数 **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.

返回 State name.

inline uint32_t **get_timeout_ms**() const

Get the configured timeout duration.

返回 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.

返回 Configured timeout action name.

inline uint32_t **get_update_interval**() const

Get the configured periodic update interval.

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

状态机

公共头文件：`#include "brookesia/lib_utils/state_machine.hpp"`

概述 *state_machine* 提供状态机实现，支持状态注册、动作触发、转换回调与运行时控制，适合管理复杂流程与多状态切换场景。

特性

- 支持添加状态与状态转换规则
- 支持指定初始状态与强制切换
- 支持转换完成回调与运行状态查询
- 可与 *TaskScheduler* 联动进行异步调度

API 参考

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

备注： 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.

参数 **state** –Shared pointer to the state implementation.
返回 `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.

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

返回 `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.

备注: 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.

参数 **config** –Configuration for the state machine.
返回 `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.

备注: 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.

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

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

返回 `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.

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

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

参数 **target_state** –State name assigned as the new current state.
返回 `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.

备注: 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.

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

inline bool **is_running**() const

Check whether the state machine has been started.

备注: This method is thread-safe.

返回 `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.

备注: This method is thread-safe.

返回 `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.

备注: This method is thread-safe.

返回 `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.

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

返回 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.

备注: This method is thread-safe.

参数 **name** –State name to look up.
返回 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.

插件系统

公共头文件: `#include "brookesia/lib_utils/plugin.hpp"`

概述 *plugin* 提供通用插件注册表与实例管理机制，用于按名称注册、发现与创建插件实例。

特性

- 模板化插件注册与查询，按名称获取实例
- 支持延迟实例化与单例插件注册
- 提供插件枚举、释放与清理接口
- 支持宏生成注册符号，保证链接可见性

API 参考

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.

模板参数 **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.

备注: 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.

参数 **name** – **[in]** Registered plugin name.
返回 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.

备注: 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.

返回 Map from plugin name to shared plugin instance.

static inline size_t **get_plugin_count**()

Get the number of registered plugin names.

返回 Total number of registered plugins.

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

Check whether a plugin name is registered.

参数 **name** – **[in]** Plugin name to test.
返回 `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.

参数 **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.

参数 **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.

模板参数 **PluginType** –Concrete plugin type being registered.
参数

- **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.

备注: 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.

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

参数

- **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);
```

参数

- **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>")
```

参数

- **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
↪")
```

参数

- **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,
```

(下页继续)

(续上页)

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

参数

- **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 辅助工具

日志系统

公共头文件：`#include "brookesia/lib_utils/log.hpp"`

概述 *log* 提供统一的日志接口与格式化能力，可在 ESP_LOG 与标准输出之间切换，并支持基于 *std::source_location* 的上下文信息。

特性

- 支持多级别日志输出（Trace/Debug/Info/Warn/Error）
- 统一的格式化接口，兼容 *boost::format* 风格占位符
- 自动提取函数名、文件名等上下文，便于定位问题
- 提供 Trace Guard 用于自动记录函数进入/退出

API 参考

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.

参数

- **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.

模板参数 **Args** –Format argument types.
参数
- **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.

返回 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.

参数
- **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.

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

返回 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.

参数 **func_name** –Raw function signature string.
返回 Trimmed function name view.

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

Extract the leaf file name from a source path.

参数 **file_path** –Raw source file path.
返回 File name view without directory components.

template<bool **Enabled**>

class **LogTraceGuard**

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

模板参数 **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.

参数

- **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`.

错误检查

公共头文件：`#include "brookesia/lib_utils/check.hpp"`

概述 *check* 提供一组统一的检查宏，用于在参数非法、表达式失败、异常抛出、错误码异常或数值越界时执行统一处理逻辑。

特性

- 覆盖常见检查场景：`nullptr`、布尔条件、*std::exception*、*esp_err_t*、范围检查
- 支持多种失败处理模式：执行代码块、*return*、*exit*、*goto*
- 与日志系统联动，可输出失败原因与上下文信息
- 可通过配置宏切换失败处理策略（无动作、错误日志、断言）

API 参考

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.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`].

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`.

参数

- **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`].

参数

- **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`].

参数

- **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`].

参数

- **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`].

参数

- **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`.

函数守卫

公共头文件：`#include "brookesia/lib_utils/function_guard.hpp"`

概述 *FunctionGuard* 提供 RAII 风格的函数守卫，用于在作用域结束时自动执行清理逻辑，降低资源释放与流程回滚的复杂度。

特性

- 作用域退出自动执行回调，可显式 *release* 取消
- 支持移动语义，便于在函数间转移守卫
- 执行回调时捕获 *std::exception*，避免异常传播破坏清理流程

API 参考

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.

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

模板参数

- **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.

参数

- **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.

描述辅助

公共头文件：`#include "brookesia/lib_utils/describe_helpers.hpp"`

概述 *describe_helpers* 基于 Boost.Describe 和 Boost.JSON 提供“对象描述 + 序列化/反序列化”能力，降低结构体、枚举与复杂类型的配置编解码成本。

特性

- 支持枚举与结构体成员反射（描述名称、成员列表、枚举值等）
- 支持常见类型 JSON 序列化/反序列化：字符串、数值、布尔、容器、可选值等
- 覆盖 *variant*、*optional*、*map*、*vector* 等复合类型检测与处理
- 提供调试友好的描述输出能力，便于日志与配置排查

API 参考

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.

参数 **fmt** –New global format.

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

Get the current process-wide default output format.

返回 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.

返回 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 调试工具

内存分析器

公共头文件：`#include "brookesia/lib_utils/memory_profiler.hpp"`

概述 *memory_profiler* 提供堆内存使用情况的采样、统计与阈值检测能力，便于运行时监控内存趋势与异常波动。

特性

- 采样当前堆内存、历史峰值与统计信息
- 支持周期性 profiling 与手动快照
- 阈值检测与信号回调（超过阈值自动通知）
- 可与 *TaskScheduler* 联动进行定时采样

API 参考

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.

备注: 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.

参数 **config** –New profiler configuration.
返回 `true` after the configuration is stored.

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

Get the current profiling configuration.

返回 Active `ProfilingConfig`.

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

Start periodic profiling with a task scheduler.

参数

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

返回 `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.

返回 `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.

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

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

Subscribe to every profiling snapshot.

参数 **slot** –Callback invoked whenever a new snapshot is produced.

返回 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.

参数

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

返回 RAII connection handle for the registered callback.

Public Static Functions

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

Get the process-wide singleton instance.

返回 Reference to the singleton profiler.

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

Capture the current memory state.

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

返回 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.

参数 **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.

线程分析器

公共头文件：`#include "brookesia/lib_utils/thread_profiler.hpp"`

概述 *thread_profiler* 提供线程（任务）运行状态与 CPU 使用情况的采样与统计能力，便于分析系统负载与异常任务。

特性

- 采样任务状态、CPU 占用与栈等信息
- 支持周期性 profiling 与手动快照
- 提供排序、过滤与阈值检测能力
- 可与 *TaskScheduler* 联动进行定时采样

API 参考

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.

备注： 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.

参数 **config** –New profiler configuration.
返回 `true` after the configuration is stored.

ProfilingConfig **get_profiling_config**() const

Get the current profiling configuration.

返回 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.

参数

- **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`.

返回 `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.

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

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

Get the latest captured profiling snapshot.

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

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

Subscribe to every generated profiling snapshot.

参数 **slot** –Callback invoked whenever a new snapshot is produced.

返回 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.

参数

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

返回 RAII connection handle for the registered callback.

Public Static Functions

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

Get the process-wide singleton instance.

返回 Reference to the singleton profiler.

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

Capture a single raw scheduler sample.

返回 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.

参数

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

返回 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.

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

参数

- **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.

参数

- **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.

参数

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

返回 `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.

备注: 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)

参数

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

返回 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.

时间分析器

公共头文件：`#include "brookesia/lib_utils/time_profiler.hpp"`

概述 *time_profiler* 提供层级化的时间分析能力，可记录作用域与事件耗时，帮助定位性能瓶颈与热点路径。

特性

- 支持作用域与事件两类计时模型
- 生成层级化统计报告，支持自定义输出格式
- 支持线程内的独立采样与统计
- 提供便捷宏快速接入分析点

警告： 当 *ESP32-P4* 开启 `CONFIG_SPIRAM_XIP_FROM_PSRAM=y` 时，`time_profiler` 功能可能导致系统崩溃。

API 参考

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()`.

参数 **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.

参数 **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.

参数 **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.

参数 **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.

返回 `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*.

返回 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.

参数 **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

硬件抽象组件

ESP-Brookesia HAL 框架由三个组件分层协作，共同完成从开发板硬件到上层业务的抽象：

- `brookesia_hal_interface`：**定义抽象接口**，上层业务只依赖此层，与具体硬件解耦
- `brookesia_hal_adaptor`：**实现抽象接口**，通过 `esp_board_manager` 读取板级配置并初始化真实外设
- `brookesia_hal_boards`：**提供板级 YAML 配置**，描述各开发板的外设拓扑、引脚与驱动参数


上层业务 / 服务 / 示例
按接口名称发现 & 调用
brookesia_hal_interface
· 设备 / 接口抽象基类
· 音频、显示、存储等 HAL 接口定义
· 全局接口注册表
实现抽象接口
brookesia_hal_adaptor
· AudioDevice / DisplayDevice / StorageDevice 板级实现
· 通过 esp_board_manager 初始化外设
· 将接口实例注册到全局表
提供 YAML 配置
brookesia_hal_boards
· 各开发板的外设拓扑（引脚、总线）
· 逻辑设备配置（音频编解码、LCD、触摸等）
· 板级 Kconfig 默认值与驱动工厂回调

备注： 自定义开发板可通过以下两种方式接入 ESP-Brookesia HAL 框架：

- **方式一（推荐）**：在 `brookesia_hal_boards` 的 `boards/` 目录下，按照 `esp_board_manager` 规范新建开发板子目录并补充配置文件，无需修改适配层代码。详见添加自定义开发板。
- **方式二（完全自定义）**：移除对 `brookesia_hal_adaptor` 和 `brookesia_hal_boards` 的依赖，直接基于 `brookesia_hal_interface` 的抽象接口编写板级初始化代码。适用于无法使用 `esp_board_manager` 的场景，但需自行维护与接口规范的兼容性。

3.1 HAL 接口

- 组件注册表：espressif/brookesia_hal_interface
- 公共头文件：`#include "brookesia/hal_interface.hpp"`

3.1.1 概述

`brookesia_hal_interface` 是 ESP-Brookesia 的硬件抽象基础组件，在「板级实现」与「上层业务或其它子系统」之间提供统一抽象，主要能力包括：

- **设备与接口模型**：用「设备」聚合硬件单元，用「接口」表达可复用能力（编解码播放/录音、显示面板/触摸/背光、存储卷等），二者职责清晰、可组合
- **插件式注册**：具体设备与接口实现通过注册表登记，运行时按名称解析，避免业务侧硬编码实现类
- **探测与生命周期**：设备先探测是否可用，再初始化；支持批量或按名称单独初始化、对称反初始化
- **全局发现**：设备可按插件名或设备逻辑名解析；接口可在全局范围内按类型枚举或按设备内名称获取
- **常用 HAL 声明**：内置音频、显示、存储等接口的抽象定义，具体行为由适配层实现

3.1.2 功能特性

设备与接口

设备表示一块可独立管理的硬件单元（例如某路音频编解码、某套显示子系统、某套存储子系统）。设备在正式工作前需要 **探测**：仅当硬件在当前环境下可用时，才进入初始化。初始化阶段，设备在内部收集需要对外暴露的 **接口实例**。

接口表示一类稳定的能力边界（例如编解码播放、录音、面板绘图、触摸采样、背光控制、存储介质与文件系统发现等）。同一抽象类型下可以存在多份实例，通过注册时使用的名称区分；名称在全局接口注册表中唯一标识一份实例。

设备与接口的实现类通过项目中的插件机制登记；框架在运行时根据名称创建或取得实例，上层只需依赖本组件中的抽象类型与约定。

注册与生命周期

批量初始化时，框架按注册表逐项处理：对每台设备依次执行探测、设备侧初始化；初始化成功后，由框架将该设备声明的接口同步登记到全局接口表。某一设备探测失败或初始化失败时，通常仅影响该设备，其余设备可继续处理。

也支持仅针对 **某一插件注册名**做初始化或反初始化。反初始化时，先撤销该设备相关接口在全局表中的登记，再执行设备自定义清理，并释放设备内部对接口的持有关系。

典型流程如下：

1. 已注册设备进入 **探测**；
2. 探测通过则 **初始化**，并将接口登记到全局注册表；探测不通过则 **跳过该设备**。

发现与命名

与设备相关的名称分两类，用途不同：

名称类型	含义
插件名	插件注册表中的键，用于按注册项直接定位设备实例
设备名	设备对象自身携带的逻辑名，可与插件名相同或不同

按其中任一路径都可以在运行时解析到对应设备。

对接口的访问面向 **全局接口注册表**：可按接口类型列出当前所有可转换的实例，也可取得某类型下首个匹配实例（名称与实例成对返回）。若已从设备对象入手，也可仅在该设备已发布的接口集合中，用登记时使用的完整接口名取回能力。

多设备并存时，建议为接口注册名增加可区分设备的前缀或其它命名空间，避免全局冲突。

内置能力范畴

头文件中提供常用 HAL 接口的抽象定义，涵盖音频编解码播放与录音、显示面板与触摸与背光、以及存储文件系统发现等。它们描述静态信息、能力参数与虚接口约定；具体寄存器操作、总线与时序由板级适配或其它组件完成。

以下接口头文件可通过 `brookesia/hal_interface/interfaces.hpp` 一次性引入，也可通过聚合入口 `brookesia/hal_interface.hpp` 同时引入设备基类：

头文件	主要类型
`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`

设备接口类 组件提供了以下接口类：

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

3.1.3 API 参考

设备 & 接口基类

设备基类

公共头文件：`#include "brookesia/hal_interface/device.hpp"`

API 参考

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.

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

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

Get the registry name of this device.

返回 *Device* name.

template<typename T> inline requires IsInterface< T > std::shared_ptr< T > get_inte

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

模板参数 **T** – *Interface* type that derives from *Interface*.
参数 **name** – **[in]** Fully-qualified interface name in the registry.
返回 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.

接口基类

公共头文件：`#include "brookesia/hal_interface/interface.hpp"`

API 参考

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.

参数 **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.

返回 *Interface* name.

音频接口类

编解码器播放接口

公共头文件：`#include "brookesia/hal_interface/audio/codec_player.hpp"`

类名：`AudioCodecPlayerIface`

API 参考

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.

参数 **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.

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

virtual void **close**() = 0

Close the playback backend.

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

Set playback volume.

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

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

Get current playback volume.

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

virtual bool **mute**() = 0

Set playback to mute.

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

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

virtual bool **unmute**() = 0

Unmute playback.

备注: 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.

返回 `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.

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

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

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

Get static playback capability information.

返回 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.

编解码器录音接口

公共头文件：`#include "brookesia/hal_interface/audio/codec_recorder.hpp"`

类名：`AudioCodecRecorderIface`

API 参考

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.

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

virtual **~AudioCodecRecorderIface**() = default

Virtual destructor for polymorphic recording interfaces.

virtual bool **open**() = 0

Open the recording backend.

返回 `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.

参数

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

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

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

Set the general gain.

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

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

Set the channel gains.

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

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

Get static recording capability information.

返回 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.

显示接口类

显示面板接口

公共头文件：`#include "brookesia/hal_interface/display/panel.hpp"`

API 参考

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.

参数 **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)`.

参数

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

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

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

Get the driver-specific data.

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

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

Get static panel capability information.

返回 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.

返回 Number of bits per pixel.

inline bool **is_valid**() const

Check if the panel information is valid.

返回 `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.

触摸接口

公共头文件：`#include "brookesia/hal_interface/display/touch.hpp"`

类名：`DisplayTouchIface`

API 参考

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.

参数 **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.

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

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

Register an interrupt handler.

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

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

Get the driver-specific data.

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

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

Get static touch capability information.

返回 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.

背光接口

公共头文件：`#include "brookesia/hal_interface/display/backlight.hpp"`

API 参考

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.

参数 **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.

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

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

Get current backlight brightness.

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

virtual bool **turn_on**() = 0

Turn on the backlight.

备注: 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.

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

virtual bool **turn_off**() = 0

Turn off the backlight.

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

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

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

Get static backlight capability information.

返回 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.

存储接口类

文件系统接口

公共头文件：`#include "brookesia/hal_interface/storage/fs.hpp"`

类名：`StorageFsIface`

API 参考

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.

返回 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 适配

- 组件注册表：espressif/brookesia_hal_adaptor
- 公共头文件：`#include "brookesia/hal_adaptor.hpp"`

3.2.1 概述

`brookesia_hal_adaptor` 是 ESP-Brookesia 的板级 HAL 适配实现，基于*HAL 接口* 的设备/接口模型，通过 `esp_board_manager` 初始化真实外设，并将 **音频**、**显示**、**存储**三类能力注册到全局 HAL 表，供上层按名称发现和使用。

3.2.2 功能特性

内置设备

本组件提供三台板级设备，每台设备以单例形式注册，初始化后将对应的接口发布到全局表：

设备类（逻辑名）	注册的接口实现	说明
`AudioDevice ("Audio")`	`AudioCodecPlayerIface (CODEC_PLAYER_IMPL_NAME)`、`AudioCodecRecorderIface (CODEC_RECORDER_IMPL_NAME)`	播放经板级 Audio DAC；录音经 Audio ADC。子实现可在 Kconfig 中独立关闭，并依赖板级能力符号 `ESP_BOARD_DEV_AUDIO_CODEC_SUPPORT`。
`DisplayDevice ("Display")`	`DisplayBacklightIface (LEDC_BACKLIGHT_IMPL_NAME)`、`DisplayPanelIface (LCD_PANEL_IMPL_NAME)`、`DisplayTouchIface (LCD_TOUCH_IMPL_NAME)`	LEDC 背光、LCD 面板、I2C 触摸可分别开关；分别依赖 `ESP_BOARD_DEV_LEDC_CTRL_SUPPORT`、`ESP_BOARD_DEV_DISPLAY_LCD_SUPPORT`、`ESP_BOARD_DEV_LCD_TOUCH_I2C_SUPPORT`。
`StorageDevice ("Storage")`	`StorageFsIface (GENERAL_FS_IMPL_NAME)`	通用文件系统实现，支持 SPIFFS（依赖 `ESP_BOARD_DEV_FS_SPIFFS_SUPPORT`）与 SD 卡（FATFS，依赖 `ESP_BOARD_DEV_FS_FAT_SUPPORT`），按 Kconfig 启用。

配置与参数

各子接口可在 `menuconfig` 的 **ESP-Brookesia: Hal Adaptor Configurations** 中独立开启或关闭；默认能力参数（音量范围、录音格式、背光亮度范围等）也可在 `menuconfig` 中调整，由 `macro_configs.h` 映射为编译宏供实现使用。

若需在初始化前覆盖默认能力参数，可在对应设备单例上调用 `set_codec_player_info`、`set_codec_recorder_info` 或 `set_ledc_backlight_info`。初始化完成后再调用通常不会生效。

3.2.3 API 参考

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.

参数 **`info`** –**[in]** Backlight capability descriptor (brightness range and default).
返回 `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.

返回 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.

参数 **info** –**[in]** Codec player capability descriptor (volume range and default).
返回 `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.

参数 **info** –**[in]** Codec recorder capability descriptor (format, channels, gains, etc.).
返回 `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.

返回 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.

返回 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 开发板支持

- 组件注册表：espressif/brookesia_hal_boards

3.3.1 概述

brookesia_hal_boards 是 ESP-Brookesia 的开发板配置集合，基于 esp_board_manager 组件以 YAML 文件描述各开发板的外设拓扑与设备参数，供*HAL 适配* 在运行时无需硬编码即可完成硬件初始化。

3.3.2 支持的开发板

板级名称	芯片	描述
esp_vocat_board_v1_0	ESP32-S3	ESP VoCat Board V1.0 —AI 宠物伴侣开发板
esp_vocat_board_v1_2	ESP32-S3	ESP VoCat Board V1.2 —AI 宠物伴侣开发板
esp_box_3	ESP32-S3	ESP-BOX-3 开发板
esp32_s3_korvo2_v3	ESP32-S3	ESP32-S3-Korvo-2 V3 开发板
esp32_p4_function_ev	ESP32-P4	ESP32-P4-Function-EV-Board
esp_sensair_shuttle	ESP32-C5	ESP Sensair Shuttle 模块

3.3.3 目录结构

每块开发板在 boards/< 板级名称>/ 目录下包含以下文件：

```
boards/
└── <board>/
    ├── board_info.yaml           # 开发板元信息（名称、芯片、版本、制造商等）
    ├── board_devices.yaml        # 逻辑设备配置（音频编解码、LCD
↪显示、触摸、存储等）
    ├── board_peripherals.yaml    # 底层外设配置（I2C/I2S/SPI 总线、GPIO、LEDC 等）
    ├── sdkconfig.defaults.board # 开发板专用 Kconfig 默认值（Flash、PSRAM 等）
    └── setup_device.c            #
↪板级设备工厂回调（用于需要自定义驱动初始化的场景）
```

设备类型

board_devices.yaml 以设备为单位描述板上的逻辑功能模块，常见设备类型如下：

设备类型	说明
audio_codec	音频编解码芯片（DAC 播放 / ADC 录音），支持 ES8311、ES7210、内置 ADC 等
display_lcd	LCD 显示屏，支持 SPI（ST77916、ILI9341）、DSI（EK79007）等接口
lcd_touch	触摸面板，支持 CST816S、GT911 等 I2C 触控芯片
ledc_ctrl	基于 LEDC 的 PWM 背光控制
fs_fat / fs_spiffs	文件系统存储，支持 SD 卡（SDMMC/SPI）和 SPIFFS
camera	摄像头（CSI 接口）
power_ctrl	GPIO 供电控制（音频电源、LCD/SD 卡电源等）
gpio_ctrl	通用 GPIO 控制（LED、按键等）

外设配置

board_peripherals.yaml 描述底层硬件资源的引脚分配与参数：

- **I2C**：SDA/SCL 引脚、端口号
- **I2S**：MCLK/BCLK/WS/DOUT/DIN 引脚、采样率、位深
- **SPI**：MOSI/MISO/CLK/CS 引脚、SPI 主机编号、传输大小
- **LEDC**：背光 GPIO、PWM 频率与分辨率
- **GPIO**：供电控制、功放使能、LED 等独立引脚配置

`sdkconfig.defaults.board` 包含与该开发板硬件强相关的 Kconfig 默认值，例如 Flash 大小、PSRAM 模式与频率、CPU 主频，以及 `brookesia_hal_adaptor` 的录音格式参数等。

若驱动需要自定义初始化流程（例如向 LCD 传入厂商特定的寄存器序列），则通过 `setup_device.c` 中的工厂回调函数实现。

3.3.4 使用方法

选择开发板

在工程根目录执行以下命令，指定目标开发板：

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

`<board>` 的可选值见支持的开发板。若 `brookesia_hal_boards` 以本地路径依赖引入，需通过 `-c` 参数指定 `boards/` 目录路径：

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

备注： 在使用了 `idf_ext.py` 的示例工程中，`-c` 参数会在构建时自动注入，无需手动添加。

添加自定义开发板

在 `boards/` 目录（或任意自定义目录）下创建新的开发板子目录，按以下顺序添加文件：

1. **board_info.yaml**：填写开发板名称、芯片型号、版本号和描述
2. **board_peripherals.yaml**：按实际引脚和总线配置填写外设参数
3. **board_devices.yaml**：按实际板载外设填写设备类型和配置
4. **sdkconfig.defaults.board**：添加与该板相关的 Kconfig 默认值
5. **setup_device.c** （可选）：如驱动需要额外初始化步骤则实现对应工厂函数

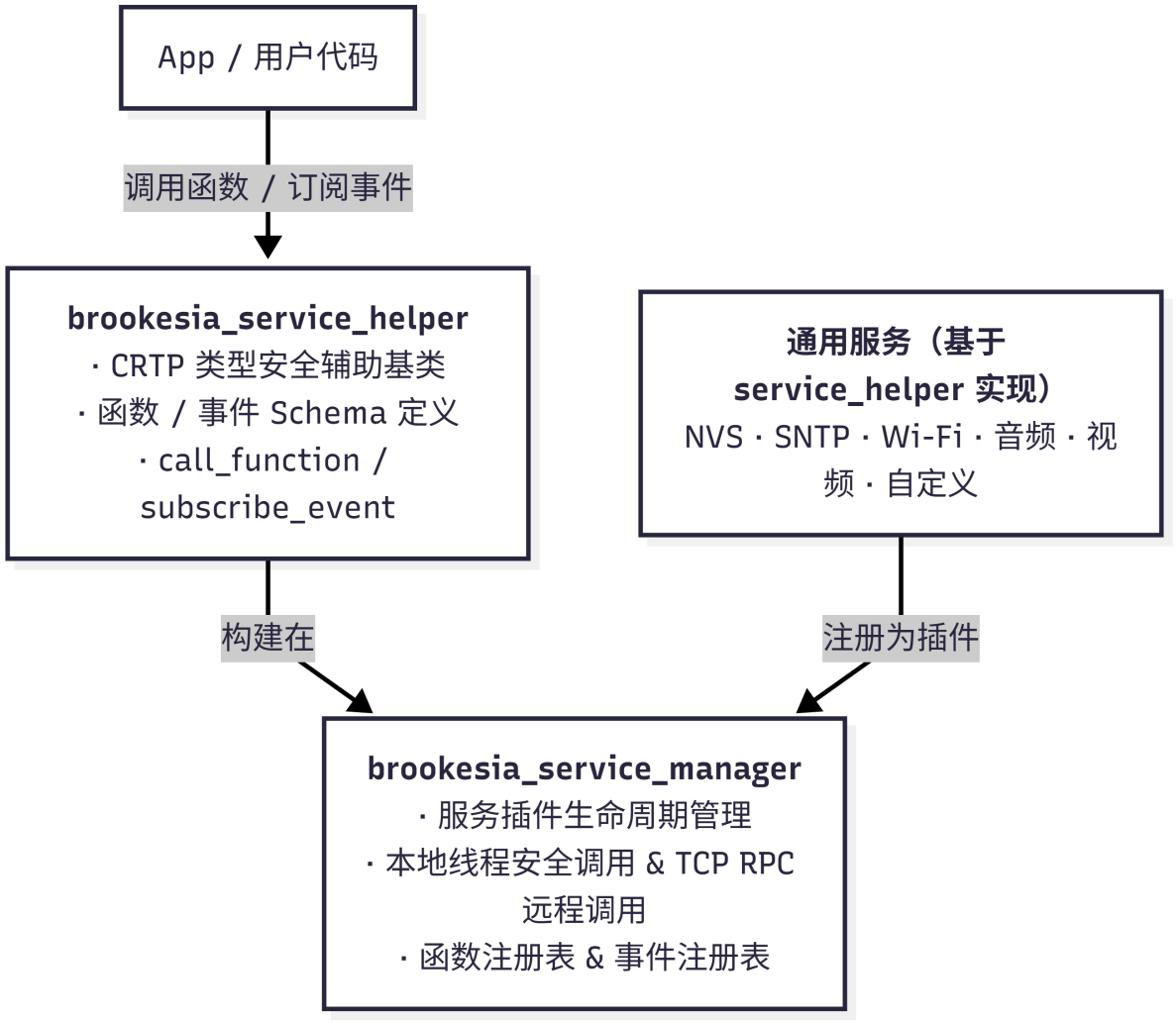
完成后执行 `idf.py gen-bmgr-config -b <new_board>` 即可使用。

备注： 关于 `esp_board_manager` 配置格式的完整说明，请参考 esp_board_manager 组件文档。

Chapter 4

服务组件

本分类包含 ESP-Brookesia 服务框架组件的说明内容。ESP-Brookesia 服务框架由服务框架层和通用服务层组成，各组件的层级关系如下：



- `brookesia_service_manager`：服务框架核心，负责服务插件注册、本地/RPC 两种通信模式下的函数路由与事件分发
- `brookesia_service_helper`：基于 CRTP 的类型安全辅助层，简化服务的函数/事件定义与调用方式

- **通用服务**：基于 `service_helper` 实现的具体业务服务，注册到 `service_manager` 后可被上层按名称发现和调用

4.1 服务框架

4.1.1 服务管理器

- 组件注册表：espressif/brookesia_service_manager
- 公共头文件：`#include "brookesia/service_manager.hpp"`

概述

brookesia_service_manager 是 ESP-Brookesia 的服务管理框架核心组件，统一提供服务生命周期管理、函数注册与事件分发、本地调用与远程 RPC 调用等能力。

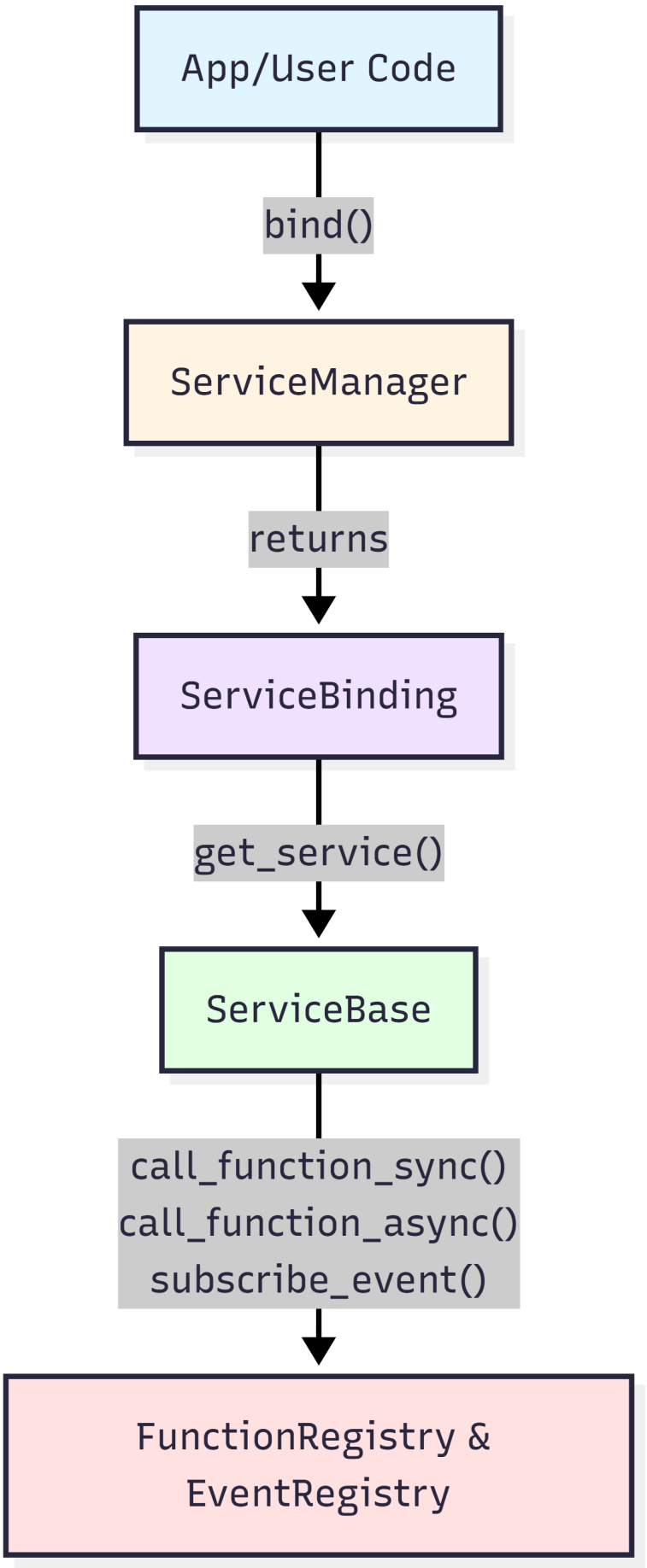
特性

- 生命周期管理：集中式管理服务初始化、启动、停止与反初始化流程。
- 双通信模式：同时支持本地高性能调用和基于 TCP/JSON 的远程 RPC。
- 统一函数与事件模型：提供函数定义、注册、分发与事件订阅机制。
- 线程安全调度：可通过本地运行器与任务调度机制实现并发安全执行。
- 解耦集成：应用代码通过统一 API 调用服务，不依赖具体服务实现细节。

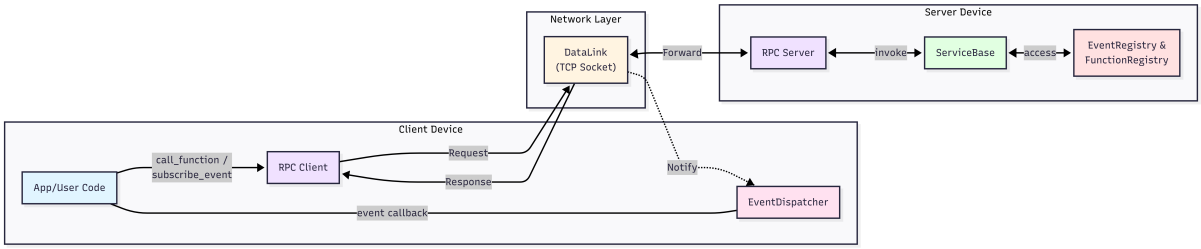
通信架构

brookesia_service_manager 支持本地模式和远程 RPC 两种通信方式，前者适合同设备内的高频调用，后者适合跨设备或跨语言场景。

本地模式 本地模式下，应用通过 *ServiceManager* 绑定服务后，直接通过 *ServiceBase* 访问函数和事件注册表，调用链路短、性能高，并且不需要网络通信。


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

远程 RPC 模式 远程 RPC 模式下，客户端通过 TCP Socket 和 JSON 协议访问远端服务，适合将服务能力暴露给其他设备或其他语言环境。



本地调用 vs 远程 RPC

对比项	本地调用（ServiceBase）	远程 RPC（rpc::Client）
部署位置	同一设备内	跨设备通信
通信方式	直接函数调用	TCP Socket + JSON
延迟	毫秒级，较低	受网络影响，通常更高
性能	无序列化开销，性能更高	需要序列化与反序列化
适用频率	高频调用	中低频调用
线程安全	具备异步调度与保护机制	依赖网络隔离
语言支持	C++	语言无关
网络依赖	不需要网络	需要同一局域网或可达网络
典型场景	设备内服务协作	跨设备或跨语言服务调用

模块介绍

服务运行层 服务运行层负责服务生命周期管理、调用与事件执行调度、依赖绑定和与 RPC 对接，核心由 `ServiceBase`、`ServiceManager` 和 `LocalTestRunner` 构成。

API 参考

函数系统 函数系统用于定义服务可调用接口、校验调用参数并分派到具体处理函数，核心包含函数定义模型和函数注册表。

API 参考

事件系统 事件系统用于描述事件结构、校验事件数据并将事件分发到本地订阅者或远程 RPC 订阅者，核心包含事件定义、注册表和调度器。

API 参考

RPC 通信 RPC 子系统通过 TCP + JSON 对外暴露服务函数与事件能力，提供协议层、数据链路层、连接桥接层及客户端/服务端实现。

API 参考

通用 通用模块定义服务管理器中复用的通用类型和宏，为其他模块提供公共基础能力。

API 参考

服务运行层

API 参考

服务基类 公共头文件：`#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.

参数 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}
            }
```

(下页继续)

(续上页)

```
        }
    };
}
```

返回 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}
            }
        }
    };
}
```

返回 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)

参数

- **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

返回 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)

参数

- **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

返回 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)

参数

- **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

返回 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)

参数

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

返回 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)

参数

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

返回 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)

参数

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

返回 FunctionResult Result of the function call

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

Subscribe to an event.

参数

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

返回 *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.

返回 true if initialized, false otherwise

inline bool **is_running**() const

Check if the service is running.

返回 true if running, false otherwise

inline bool **is_server_connected**() const

Check if the service is connected to server.

返回 true if connected, false otherwise

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

Get the service attributes.

返回 const *Attributes*& Reference to service attributes

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

Get the call task group name.

返回 std::string Call task group name

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

Get the event task group name.

返回 std::string Event task group name

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

Get the request task group name.

返回 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.

模板参数 **T** –Return value type, can be void or any type convertible to FunctionValue
参数 **result** –**[in]** std::expected object
返回 FunctionResult Converted result

struct **Attributes**

Service attributes configuration.

Public Functions

inline bool **has_scheduler**() const

Check whether a dedicated task scheduler configuration is present.

返回 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.

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

返回 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)

服务管理器 公共头文件: `#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.

返回 true if valid and service is running, false otherwise

inline explicit **operator bool**() const

Explicit conversion to bool.

返回 true if valid, false otherwise

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

Get the service object.

返回 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.

参数 **name** –**[in]** Dependency service name
返回 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.

返回 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.

参数 **config** –**[in]** Task scheduler start configuration
返回 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.

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

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

Remove a service by name.

参数 **name** –**[in]** Service name
返回 true if removed successfully, false otherwise

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

Bind a service by name.

参数 **name** –**[in]** Service name
返回 *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.

参数

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

返回 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.

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

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

Disconnect RPC server from services.

参数 **names** –**[in]** Service names to disconnect (empty means all services)
返回 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.

参数 **config** –**[in]** RPC client configuration
返回 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.

参数

- **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)

返回 FunctionResult Result of the function call

inline bool **is_initialized**() const

Check if the service manager is initialized.

返回 true if initialized, false otherwise

inline bool **is_running**() const

Check if the service manager is running.

返回 true if running, false otherwise

inline bool **is_rpc_server_running**() const

Check if the RPC server is running.

返回 true if running, false otherwise

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

Get a service by name.

参数 **name** –**[in]** Service name
返回 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.

返回 *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.

服务本地运行器 公共头文件: `#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.

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

返回 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.

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

返回 true if all tests passed, false otherwise

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

Get test results.

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

struct **RunTestsConfig**

函数系统

API 参考

函数定义 公共头文件: `#include "brookesia/service_manager/function/definition.hpp"`

Header File

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

函数注册表 公共头文件：`#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.

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

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

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

Remove a registered function.

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

bool **remove_all**()

Remove every registered function.

返回 true if the registry was cleared successfully.

FunctionResult **call**(const std::string &func_name, FunctionParameterMap parameters)

Validate parameters and invoke a registered function.

参数
- **func_name** –**[in]** Function name to call.
- **parameters** –**[in]** Parameter map passed to the handler.

返回 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.

参数 **func_name** –**[in]** Function name to query.
返回 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.

返回 std::vector<FunctionSchema> Copy of the registered schemas.

boost::json::array **get_schemas_json**()

Export all registered function schemas as JSON.

返回 boost::json::array JSON representation of every schema.

inline bool **has**(const std::string &func_name)

Check whether a function is registered.

参数 **func_name** –**[in]** Function name to check.
返回 true if the function exists.

inline size_t **get_count**()

Get the number of registered functions.

返回 size_t Count of registered functions.

事件系统

API 参考 公共头文件: `#include "brookesia/service_manager/event/definition.hpp"`

Header File

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

事件调度器 公共头文件: `#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.

参数
- **subscription_id** –**[in]** Subscription id returned by the RPC server.
- **callback** –**[in]** Callback to invoke when matching notifications arrive.

返回 true on success, false if the subscription id is already registered.

void **unsubscribe**(const std::string &subscription_id)

Remove a previously registered subscription callback.

参数 **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.

参数

- **subscription_ids** –**[in]** Subscription ids targeted by the notification.
- **event_items** –**[in]** Event payload associated with the notification.

事件注册表 公共头文件: `#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.

参数 **event_schema** –**[in]** Schema to add.
返回 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.

参数 **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.

参数

- **event_name** –**[in]** Event name to validate against.
- **event_items** –**[in]** Event payload to validate.

返回 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.

参数

- **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.

返回 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.

参数 **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.

参数 **subscriptions** –**[in]** Subscription ids to remove.

inline const EventSchema ***get_schema**(const std::string &event_name)

Look up the schema for a registered event.

参数 **event_name** –**[in]** Event name to query.
返回 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.

返回 std::vector<EventSchema> Copy of the registered schemas.

boost::json::array **get_schemas_json**()

Export all registered event schemas as JSON.

返回 boost::json::array JSON representation of every schema.

inline bool **has**(const std::string &event_name)

Check whether an event schema exists.

参数 **event_name** –**[in]** Event name to check.
返回 true if the event is registered.

inline size_t **get_count**()

Get the number of registered event schemas.

返回 size_t Count of registered events.

Subscriptions **get_subscriptions**(const std::string &event_name)

Get all RPC subscription ids associated with an event.

参数 **event_name** –**[in]** Event name to query.
返回 Subscriptions Copy of the registered subscription ids.

Signal ***get_signal**(const std::string &event_name)

Get the in-process signal associated with an event.

参数 **event_name** –**[in]** Event name to query.
返回 Signal* Pointer to the signal, or `nullptr` if the event does not exist.

RPC 通信

API 参考

RPC 协议 公共头文件：`#include "brookesia/service_manager/rpc/protocol.hpp"`

Header File

- service/brookesia_service_manager/include/brookesia/service_manager/rpc/protocol.hpp

RPC 连接 公共头文件：`#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.

参数

- **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.

参数 **responder** –**[in]** Transport callback.

inline void **set_notifier**(*Notifier* notifier)

Set the notifier used for sending RPC notifications.

参数 **notifier** –**[in]** Transport callback.

inline void **set_request_handler**(*RequestHandler* request_handler)

Set a custom request handler.

参数 **request_handler** –**[in]** Custom transport-aware request handler.

inline void **activate**(bool active)

Enable or disable request handling on this connection.

参数 **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.

参数

- **request_id** –**[in]** RPC request id.
- **connection_id** –**[in]** Transport connection id.
- **method** –**[in]** Requested method name.
- **parameters** –**[in]** Method parameters.

返回 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.

参数 **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.

参数

- **event_name** –**[in]** Event name to publish.
- **event_items** –**[in]** Event payload.

返回 true if notifications were dispatched successfully.

bool **respond_request**(size_t connection_id, Response &&response)

Send an RPC response to one client connection.

参数

- **connection_id** –**[in]** Transport connection id.
- **response** –**[in]** Response to send.

返回 true if the response was handled to the transport.

RPC 数据链路基础 公共头文件: `#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.

参数 **executor** –**[in]** Executor that owns asynchronous I/O operations.

inline void **set_on_data_received**(*OnDataReceived* callback)

Set the callback used for incoming payloads.

参数 **callback** –**[in]** Callback to install.

inline void **set_on_connection_established**(*OnConnectionEstablished* callback)

Set the callback used for successful connection establishment.

参数 **callback** –**[in]** Callback to install.

inline void **set_on_connection_closed**(*OnConnectionClosed* callback)

Set the callback used for connection teardown.

参数 **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.

返回 size_t Current global socket count.

static inline size_t **get_max_global_sockets_count**()

Get the configured global socket cap.

返回 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.

返回 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.

返回 size_t Historical peak socket count.

RPC 数据链路客户端 公共头文件: `#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.

参数 **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.

参数

- **host** –**[in]** Remote host name or address.
- **port** –**[in]** Remote TCP port.
- **timeout_ms** –**[in]** Connection timeout in milliseconds.

返回 true if the connection succeeds.

bool **disconnect**()

Disconnect the active client connection.

返回 true if a connection existed and cleanup succeeded.

bool **send_data**(std::string &&data)

Send one payload over the active connection.

参数 **data** –**[in]** Framed payload to send.
返回 true if the send operation was queued successfully.

bool **is_connected**()

Check whether the client currently has an active connection.

返回 true if connected.

RPC 数据链路服务端 公共头文件: `#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.

参数

- **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.

参数

- **port** –**[in]** Local TCP port.
- **timeout_ms** –**[in]** Startup timeout in milliseconds.

返回 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.

参数

- **connection_id** –**[in]** *Client* connection id.
- **data** –**[in]** Framed payload to send.

返回 true if the payload was queued successfully.

size_t **get_active_connections_count**()

Get the number of currently active client connections.

返回 size_t Active connection count.

std::vector<size_t> **get_active_connection_ids**()

Get the ids of all active client connections.

返回 std::vector<size_t> Snapshot of active connection ids.

inline bool **is_running**()

Check whether the server transport is currently accepting traffic.

返回 true if running.

inline size_t **get_max_connections_count**()

Get the configured maximum number of client connections.

返回 size_t Connection cap.

inline bool **is_connection_limit_reached**()

Check whether the local connection cap has been reached.

返回 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.

返回 size_t Historical maximum for this server instance.

RPC 客户端 公共头文件：`#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.

参数 **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.

参数

- **executor** –**[in]** Executor used for async socket operations.
- **on_disconnect_callback** –**[in]** Callback invoked if the transport disconnects.

返回 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.

参数

- **host** –**[in]** Remote host name or address.
- **port** –**[in]** Remote port.
- **timeout_ms** –**[in]** Connection timeout in milliseconds.

返回 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.

参数

- **target** –**[in]** Target service name.
- **method** –**[in]** Remote function name.
- **params** –**[in]** JSON object containing function parameters.

返回 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.

参数

- **target** –**[in]** Target service name.
- **method** –**[in]** Remote function name.
- **params** –**[in]** JSON object containing function parameters.
- **timeout_ms** –**[in]** Wait timeout in milliseconds.

返回 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.

参数

- **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.

返回 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.

参数

- **target** –**[in]** Target service name.
- **subscription_ids** –**[in]** Subscription ids to cancel.
- **timeout_ms** –**[in]** RPC timeout in milliseconds.

返回 true if the unsubscribe request succeeds.

RPC 服务端 公共头文件：`#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.

参数

- **executor** –**[in]** Executor used for socket operations.
- **config** –**[in]** Transport configuration.

~Server()

Destructor.

bool **init**()

Initialize internal transport resources.

返回 true if initialization succeeds.

void **deinit**()

Deinitialize the server and release transport resources.

bool **start**(uint32_t timeout_ms)

Start listening for RPC clients.

参数 **timeout_ms** –**[in]** Startup timeout in milliseconds.
返回 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.

参数 **connection** –**[in]** Service connection wrapper to expose.
返回 true if the connection is added successfully.

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

Remove a service connection by service name.

参数 **name** –**[in]** Service name to remove.
返回 true if a matching connection was removed.

std::shared_ptr<*ServerConnection*> **get_connection**(const std::string &name)

Get a registered service connection.

参数 **name** –**[in]** Service name to query.
返回 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.

通用

API 参考 公共头文件：`#include "brookesia/service_manager/common.hpp"`

Header File

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

4.1.2 服务辅助

- 组件注册表：espressif/brookesia_service_helper
- 公共头文件：`#include "brookesia/service_helper.hpp"`

概述

brookesia_service_helper 是 ESP-Brookesia 服务层面向应用的统一辅助接口组件，提供各类通用服务的 helper 包装与契约入口。

特性

- 统一入口：通过一组 helper 类封装不同服务能力，降低业务层接入复杂度。
- 契约友好：与函数/事件 schema 体系配套，便于进行标准化调用与文档生成。
- 组件解耦：应用面向 helper 编程，减少对具体 service provider 实现的直接依赖。
- 可扩展：支持为新增服务能力持续扩展 helper 子模块与对应 API 文档。

模块介绍

基类

- 公共头文件：`#include "brookesia/service_helper/base.hpp"`

概述 *esp_brookesia::service::helper::Base<Derived>* 是 Service Helper 体系的 CRTP 基类，为各具体 helper 提供统一的函数调用、事件订阅与 schema 校验能力。

特性

- CRTP 契约约束：通过 *DerivedMeta* 要求派生类提供 *FunctionId/EventId/get_name/get_*_schemas* 等标准元信息。
- 统一函数调用：提供 *call_function_sync* 与 *call_function_async*，自动完成参数打包与返回值解析。
- 类型安全事件订阅：提供多种 *subscribe_event* 重载，支持按回调签名自动适配事件参数。
- 服务可用性检查：通过 *is_available*、*get_function_schema*、*get_event_schema* 快速判断和获取能力描述。
- 超时控制：同步调用支持在参数末尾附加 *Timeout(ms)*，用于控制等待时长。
- 回调适配宏：提供 *BROOKESIA_SERVICE_HELPER_FUNC_HANDLER_**，简化服务函数注册时的参数转换样板代码。

使用方式 派生 helper 需定义函数/事件枚举和 schema，并继承 *Base<Derived>* 复用通用调用逻辑。

```
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 参考

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.

模板参数 **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;
    });
```

备注： REQUIRED: First parameter must be `std::string` or `std::string_view` to receive `event_name`.

备注： Second parameter must not be `EventItemMap`; use the other overload for that case.

备注： Remaining parameters are extracted from `EventItemMap` based on the event-schema order.

备注： All non-`event_name` parameter types must satisfy the `ConvertibleToEventItem` concept.

备注： Parameter types must match the actual `EventItem` types in the map.

模板参数

- **`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.

返回 *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.

参数 **`result`** –**[in]** Function result from service call
返回 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));
```

备注： Arguments are packed into std::vector<FunctionValue> in the order provided

备注： Last argument can be Timeout(milliseconds) to specify custom timeout

备注： All non-Timeout arguments must satisfy `ConvertibleToFunctionValue` concept

模板参数

- **`ReturnType`** –Expected return type (default: void)
- **`FunctionIdType`** –Type of the function identifier (enum)
- **`Args`** –Types of function arguments (must be convertible to FunctionValue)

参数

- **`function_id`** –Function identifier
- **`args`** –Function arguments in order, last argument can be Timeout(ms)

返回 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
});
```

备注： Arguments are packed into std::vector<FunctionValue> in the order provided

备注： All arguments (except optional FunctionResultHandler) must satisfy ConvertibleToFunctionValue concept

备注： This overload is not used when the first argument is FunctionParameterMap or boost::json::object

模板参数
- **FunctionIdType** –Type of the function identifier (enum)
- **Args** –Types of function arguments (must be convertible to FunctionValue, optionally with FunctionResultHandler at the end)

参数
- **function_id** –Function identifier
- **args** –Function arguments in order, optionally with FunctionResultHandler at the end

返回 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
    });
```

模板参数 **EventIdType** –Type of the event identifier (enum)

参数
- **event_id** –Event identifier
- **slot** –Signal slot function with signature: void(const std::string &event_name, const EventItemMap &event_items)

返回 *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.

模板参数 **T** –Return value type, can be void or any type convertible to FunctionValue
参数 **result** –**[in]** std::expected object
返回 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();
```

模板参数 **EventIdValue** –The specific EventId enum value to monitor

Public Functions

inline bool **start**()

Start monitoring for events.
返回 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.
参数
- **expected_items** –The expected items to find
- **timeout_ms** –Maximum time to wait in milliseconds

返回 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.
参数 **timeout_ms** –Maximum time to wait in milliseconds
返回 true if any event was received, false on timeout

inline size_t **get_count**() const

Get the number of received events.
返回 Number of events received since *start()* or last *clear()*

inline bool **has**(const ReceivedItmes &expected_items) const

Check if specific items have been received.
参数 **expected_items** –The expected items to find
返回 true if matching items exists in received items

inline const std::vector<ReceivedItmes> &**get_all**() const

Get all received events (unfiltered)
返回 Copy of all received events

inline std::optional<ReceivedItmes> **get_last**() const

Get the last received event.

返回 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
}
```

模板参数 **Args** –Expected types for each position in the event (bool, double, std::string, boost::json::object, boost::json::array, RawBuffer)

返回 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());
}
```

模板参数 **Args** –Expected types for each position in the event (bool, double, std::string, boost::json::object, boost::json::array, RawBuffer)

返回 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.

返回 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())*

参数

- **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))*

参数

- **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))

参数

- **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))

参数

- **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

音频 Helper

- 公共头文件：`#include "brookesia/service_helper/audio.hpp"`

概述 本页用于查看 Audio helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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*.

返回 std::string_view Stable service name.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get function schemas exported by audio service.

返回 std::span<const FunctionSchema> Static function schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas exported by audio service.

返回 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

表情 Helper

- 公共头文件：`#include "brookesia/service_helper/expression/emote.hpp"`

概述 本页用于查看 Expression/Emote helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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

- 公共头文件：`#include "brookesia/service_helper/nvs.hpp"`

概述 本页用于查看 NVS helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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*.

返回 std::string_view Stable service name.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get function schemas exported by *NVS* service.

返回 std::span<const FunctionSchema> Static function schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas exported by *NVS* service.

返回 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

备注: The storage method depends on the type T:

模板参数 **T** –The type of the value to save
参数

- **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

返回 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

---

备注: The retrieval method depends on the type T and matches the storage method used in *save_key_value()*:

模板参数 **T** –The type of the value to retrieve

参数

- **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

返回 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.

参数

- **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

返回 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

- 公共头文件：`#include "brookesia/service_helper/sntp.hpp"`

概述 本页用于查看 SNTP helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 std::string_view Stable service name.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get the function schemas exported by the *SNTP* service.

返回 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.

返回 std::span<const EventSchema> Empty span because *SNTP* exposes no events.

视频 Helper

- 公共头文件：`#include "brookesia/service_helper/video.hpp"`

概述 本页用于查看 Video helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_encoder_event_schemas**()

Get all encoder event schemas.

返回 std::span<const EventSchema> Static schema span.

static inline std::span<const FunctionSchema> **get_decoder_function_schemas**()

Get all decoder function schemas.

返回 std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_decoder_event_schemas**()

Get all decoder event schemas.

返回 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.

返回 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.

返回 std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas for this encoder instance.

返回 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.

返回 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.

返回 std::span<const FunctionSchema> Static schema span.

static inline std::span<const EventSchema> **get_event_schemas**()

Get event schemas for this decoder instance.

返回 std::span<const EventSchema> Static schema span.

Wi-Fi Helper

- 公共头文件：`#include "brookesia/service_helper/wifi.hpp"`

概述 本页用于查看 Wi-Fi helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 Constant service name string.

static inline std::span<const FunctionSchema> **get_function_schemas**()

Get all function schemas exposed by this helper.

返回 Read-only span of function schemas.

static inline std::span<const EventSchema> **get_event_schemas**()

Get all event schemas exposed by this helper.

返回 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.

参数 **rssi** – **[in]** RSSI value in dBm.

返回 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 通用服务

4.2.1 NVS

- 组件注册表：espressif/brookesia_service_nvs
- 辅助头文件：`#include "brookesia/service_helper/nvs.hpp"`
- 辅助类：`esp_brookesia::service::helper::NVS`

概述

brookesia_service_nvs 是为 ESP-Brookesia 生态系统提供的 NVS（Non-Volatile Storage，非易失性存储）服务，提供：

- **命名空间管理**：基于命名空间的键值对存储，支持多个独立的存储空间。
- **数据类型支持**：支持 布尔值、整数、字符串三种基本数据类型。
- **持久化存储**：数据存储在 NVS 分区中，断电后数据不丢失。
- **线程安全**：可选基于 *TaskScheduler* 实现异步任务调度，保证线程安全。
- **灵活查询**：支持列出命名空间中的所有键值对，或按需获取指定键的值。

功能特性

命名空间管理

- **默认命名空间**：如果不指定命名空间，将使用默认命名空间 `"storage"`。
- **多命名空间**：可以创建多个命名空间来组织不同类型的数据。
- **命名空间隔离**：不同命名空间的数据相互独立，互不影响。

支持的数据类型

- **布尔值（Bool）**：`true` 或 `false`。
- **整数（Int）**：32 位有符号整数。
- **字符串（String）**：UTF-8 编码的字符串。

备 注: 除了上述三种基本数据类型外，NVS Helper 提供的*esp_brookesia::service::helper::NVS::save_key_value()* 和*esp_brookesia::service::helper::NVS::get_key_value()* 模板助手函数还支持存储和获取任意类型的数据。

核心功能

- **列出键信息**：列出命名空间中所有键的信息（包括键名、类型等）。
- **设置键值对**：支持批量设置多个键值对。
- **获取键值对**：支持获取指定键的值，或获取命名空间中的所有键值对。
- **删除键值对**：支持删除指定键，或清空整个命名空间。

服务接口

函数

`List`

描述 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” }]

执行要求

- 是否需要调度器: 需要

参数

- `Nspace`
  - **类型**: `String`
  - **是否必填**: 可选
  - **默认值**: `"storage"`
  - **描述**: Namespace to list (optional). Uses the default namespace when omitted.

Schema JSON

CLI 命令

```
svc_call NVS List {"Nspace":null}
```

Set

描述 Set key-value pairs in an NVS namespace.

执行要求

- 是否需要调度器: 需要

参数

- `Nspace`
  - **类型**: `String`
  - **是否必填**: 可选
  - **默认值**: `"storage"`
  - **描述**: Namespace to write (optional). Uses the default namespace when omitted or empty.
- `KeyValuePairs`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: Key-value pairs as a JSON object. Allowed value types: [“Bool” ,” Int” ,” String”]. Example: { “key1” :” value1” ,” key2” :2,” key3” :true}

Schema JSON

CLI 命令

```
svc_call NVS Set {"Nspace":null,"KeyValuePairs":null}
```

Get

描述 Get key-value pairs by keys from an NVS namespace. Return type: JSON object. Example: { “key1” :” value1” ,” key2” :2,” key3” :true}

执行要求

- 是否需要调度器: 需要

参数

- `Nspace`
  - **类型**: `String`
  - **是否必填**: 可选
  - **默认值**: `"storage"`
  - **描述**: Namespace to read (optional). Uses the default namespace when omitted.
- `Keys`
  - **类型**: `Array`
  - **是否必填**: 可选
  - **默认值**: `[]`
  - **描述**: Keys to read as JSON array<string> (optional). Returns all pairs when omitted. Example: [“key1” ,” key2” ,” key3”]

Schema JSON

CLI 命令

```
svc_call NVS Get {"Nspace":null,"Keys":null}
```

Erase

描述 Erase key-value pairs from an NVS namespace.

执行要求

- 是否需要调度器: 需要

参数

- `Nspace`
  - **类型**: `String`
  - **是否必填**: 可选
  - **默认值**: `"storage"`
  - **描述**: Namespace to erase (optional). Uses the default namespace when omitted.
- `Keys`
  - **类型**: `Array`
  - **是否必填**: 可选
  - **默认值**: `[]`
  - **描述**: Keys to erase as JSON array<string> (optional). Erases all pairs when omitted or empty. Example: [“key1” ,” key2” ,” key3”]

Schema JSON

CLI 命令

```
svc_call NVS Erase {"Nspace":null,"Keys":null}
```

事件 无。

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

4.2.2 SNTP

- 组件注册表：espressif/brookesia_service_sntp
- 辅助头文件：`#include "brookesia/service_helper/sntp.hpp"`
- 辅助类：`esp_brookesia::service::helper::SNTP`

概述

brookesia_service_sntp 是为 ESP-Brookesia 生态系统提供的 SNTP（Simple Network Time Protocol，简单网络时间协议）服务，提供：

- **NTP 服务器管理**：支持配置多个 NTP 服务器，自动从服务器列表中选择可用的服务器进行时间同步。
- **时区设置**：支持设置系统时区，自动应用时区偏移。
- **自动时间同步**：自动检测网络连接状态，在网络可用时自动启动时间同步。
- **状态查询**：支持查询时间同步状态、当前配置的服务器列表和时区信息。
- **持久化存储**：可选搭配 *brookesia_service_nvs* 服务持久化保存 NTP 服务器列表和时区信息。

功能特性

NTP 服务器管理

- **默认服务器**：默认使用 `"pool.ntp.org"` 作为 NTP 服务器。
- **多服务器支持**：可以配置多个 NTP 服务器，服务会自动选择可用的服务器。
- **服务器列表**：支持获取当前配置的所有 NTP 服务器列表。

时区设置

- **默认时区**：默认时区为 CST-8（中国标准时间，UTC+8）。
- **时区格式**：支持标准的时区字符串格式（如 `UTC`、`CST-8`、`EST-5` 等）。
- **自动应用**：设置时区后会自动应用到系统时间。

核心功能

- **设置 NTP 服务器**：支持设置一个或多个 NTP 服务器。
- **设置时区**：支持设置系统时区。
- **启动服务**：启动 SNTP 服务，开始时间同步。
- **停止服务**：停止 SNTP 服务，停止时间同步。
- **获取服务器列表**：获取当前配置的 NTP 服务器列表。
- **获取时区**：获取当前配置的时区。
- **检查同步状态**：检查系统时间是否已与 NTP 服务器同步。
- **重置数据**：重置所有配置数据到默认值。

自动管理

- **自动加载配置**：服务启动时自动从 NVS 加载保存的配置。
- **自动保存配置**：配置更改后自动保存到 NVS。
- **网络检测**：自动检测网络连接状态，在网络可用时自动启动时间同步。
- **状态监控**：自动监控时间同步状态，更新同步标志。

服务接口

函数

SetServers

描述 Set NTP servers.

执行要求

- 是否需要调度器: 不需要

参数

- `Servers`
  - **类型**: `Array`
  - **是否必填**: 必填
  - **描述**: NTP servers as JSON array<string>. Example: [“pool.ntp.org” ,” cn.pool.ntp.org”]

Schema JSON

CLI 命令

```
svc_call SNTP SetServers {"Servers":null}
```

SetTimezone

描述 Set timezone.

执行要求

- 是否需要调度器: 不需要

参数

- `Timezone`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Timezone string.

Schema JSON

CLI 命令

```
svc_call SNTP SetTimezone {"Timezone":null}
```

Start

描述 Start SNTP service.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call SNTP Start
```

Stop

描述 Stop SNTP service.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call SNTP Stop
```

GetServers

描述 Get NTP servers. Return type: JSON array<string>. Example: [“pool.ntp.org” ,” cn.pool.ntp.org”]

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call SNTP GetServers
```

GetTimezone

描述 Get timezone. Return type: string. Example: “CST-8”

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call SNTP GetTimezone
```

IsTimeSynced

描述 Check whether time is synced. Return type: boolean. Example: true

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call SNTP IsTimeSynced
```

ResetData

描述 Reset NTP servers, timezone, and sync status.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call SNTP ResetData
```

事件 无。

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

4.2.3 Wi-Fi

- 组件注册表：espressif/brookesia_service_wifi
- 辅助头文件：`#include "brookesia/service_helper/wifi.hpp"`
- 辅助类：`esp_brookesia::service::helper::Wifi`

概述

brookesia_service_wifi 是为 ESP-Brookesia 生态系统提供的 WiFi 连接管理服务，提供：

- **状态机管理**：通过状态机统一管理 WiFi 的初始化、启动、连接等生命周期状态
- **自动重连**：支持自动连接历史 AP，并在断开后自动尝试重连
- **WiFi 扫描**：支持周期性扫描周围 AP，并自动发现可连接的 AP
- **连接管理**：管理目标 AP 和已连接 AP 列表，支持多 AP 历史记录
- **事件通知**：提供丰富的事件通知机制，实时反馈 WiFi 状态变化
- **持久化存储**：可选搭配 *brookesia_service_nvs* 服务持久化保存连接配置和其他参数

功能特性

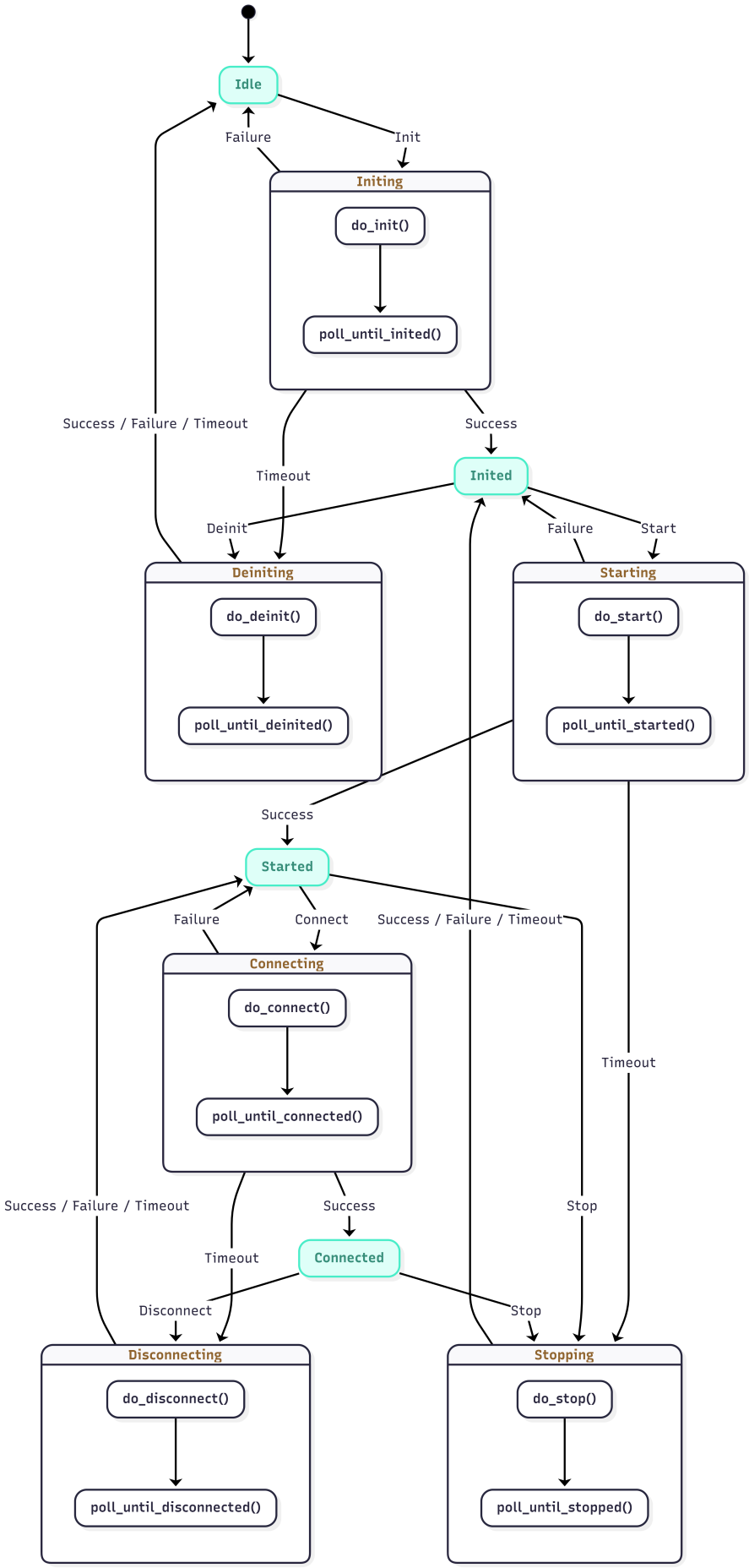
状态机管理 WiFi 服务通过状态机统一管理 WiFi 的生命周期状态，确保状态转换的安全性和一致性。状态机包含 4 个核心状态：

状态	说明
`Idle`	WiFi 未初始化，系统初始状态
`Inited`	WiFi 已初始化，但未启动，可以配置参数
`Started`	WiFi 已启动，正在扫描或等待连接
`Connected`	WiFi 已成功连接到 AP，可以正常通信

状态转换通过触发相应的动作（Action）来实现：

- **正向流程**：Idle → Inited（Init）→ Started（Start）→ Connected（Connect）
- **断开连接**：Connected → Started（Disconnect）
- **停止流程**：Started / Connected → Inited（Stop）
- **反初始化**：Inited → Idle（Deinit）

状态转换图如下：


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()

自动重连机制

- **启动时自动连接**：WiFi 启动后自动尝试连接历史可连接的 AP
- **断开后自动重连**：检测到意外断开后，自动尝试连接历史可连接的 AP
- **扫描发现自动连接**：扫描过程中发现目标 AP 或历史可连接 AP 时，自动触发连接

WiFi 扫描

- **周期性扫描**：支持配置扫描间隔和超时时间
- **扫描结果通知**：通过事件实时通知扫描到的 AP 信息
- **AP 信息**：包含 SSID、信号强度等级、是否加密等信息

SoftAP 功能

- **参数设置**：支持设置 SoftAP 的 SSID、密码、最大连接数和通道
- **最优通道选择**：如果未设置通道，则会自动扫描附近 AP 并选择最佳通道
- **配网功能**：支持启动 SoftAP 配网功能

服务接口

函数

`TriggerGeneralAction`

描述 Trigger a Wi-Fi general action.

执行要求

- 是否需要调度器: 需要

参数

- `Action`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: General action. Allowed values: [Init, Deinit, Start, Stop, Connect, Disconnect]

Schema JSON

CLI 命令

```
svc_call Wifi TriggerGeneralAction {"Action":null}
```

`GetGeneralState`

描述 Get current general state. Return type: string. Allowed values: [Idle, Initing, Inited, Deiniting, Starting, Started, Stopping, Connecting, Connected, Disconnecting]. Example: "Connected"

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi GetGeneralState
```

SetConnectAp

描述 Set target AP credentials.

执行要求

- 是否需要调度器: 需要

参数

- SSID
  - **类型**: String
  - **是否必填**: 必填
  - **描述**: AP SSID.
- Password
  - **类型**: String
  - **是否必填**: 可选
  - **默认值**: ""
  - **描述**: AP password (optional).

Schema JSON

CLI 命令

```
svc_call Wifi SetConnectAp {"SSID":null,"Password":null}
```

GetConnectAp

描述 Get target AP. Return type: JSON object. Example: { “ssid” :” ssid1” ,” password” :” password1” ,” is_connectable” :true}

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi GetConnectAp
```

GetConnectedAps

描述 Get connected AP list. Return type: JSON array<object>. Example: [{ “ssid” :” ssid1” ,” password” :” password1” ,” is_connectable” :true},{ “ssid” :” ssid2” ,” password” :” password2” ,” is_connectable” :true}]

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi GetConnectedAps
```

SetScanParams

描述 Set scan parameters.

执行要求

- 是否需要调度器: 需要

参数

- `Param`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: Scan parameters as a JSON object. Example: { “ap_count” :20,” interval_ms” :10000,” timeout_ms” :60000}

Schema JSON

CLI 命令

```
svc_call Wifi SetScanParams {"Param":null}
```

TriggerScanStart

描述 Start Wi-Fi scan.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi TriggerScanStart
```

`TriggerScanStop`

描述 Stop Wi-Fi scan.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi TriggerScanStop
```

`SetSoftApParams`

描述 Set SoftAP parameters.

执行要求

- 是否需要调度器: 需要

参数

- `Param`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: SoftAP parameters as a JSON object. Example: { “ssid” :” ssid” ,” password” :” password” ,” max_connection” :4,” channel” :1}

Schema JSON

CLI 命令

```
svc_call Wifi SetSoftApParams {"Param":null}
```

GetSoftApParams

描述 Get SoftAP parameters. Return type: JSON object. Example: {“ssid”:””,”password”:””,”max_connection”:4,” channel” :null}

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi GetSoftApParams
```

TriggerSoftApStart

描述 Start SoftAP.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi TriggerSoftApStart
```

TriggerSoftApStop

描述 Stop SoftAP.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi TriggerSoftApStop
```

TriggerSoftApProvisionStart

描述 Start SoftAP provisioning.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi TriggerSoftApProvisionStart
```

TriggerSoftApProvisionStop

描述 Stop SoftAP provisioning.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi TriggerSoftApProvisionStop
```

ResetData

描述 Reset Wi-Fi data, including target AP, scan parameters, and connected APs. Also clears NVS data.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Wifi ResetData
```

事件

`GeneralActionTriggered`

描述 Emitted when a general action is triggered.

执行要求

- 是否需要调度器: 需要

参数

- `Action`
  - **类型**: `String`
  - **描述**: General action. Allowed values: [Init, Deinit, Start, Stop, Connect, Disconnect]

Schema JSON

CLI 命令

```
svc_subscribe Wifi GeneralActionTriggered
```

`GeneralEventHappened`

描述 Emitted when a general event occurs.

执行要求

- 是否需要调度器: 需要

参数

- `Event`
  - **类型**: `String`
  - **描述**: General event. Allowed values: [Deinited, Inited, Stopped, Started, Disconnected, Connected]
- `IsUnexpected`
  - **类型**: `Boolean`
  - **描述**: Whether the event was unexpected.

Schema JSON

CLI 命令

```
svc_subscribe Wifi GeneralEventHappened
```

ScanStateChanged

描述 Emitted when scan state changes.

执行要求

- 是否需要调度器: 需要

参数

- `IsRunning`
  - **类型**: `Boolean`
  - **描述**: Whether scanning is running.

Schema JSON

CLI 命令

```
svc_subscribe Wifi ScanStateChanged
```

ScanApInfosUpdated

描述 Emitted when scan AP list is updated.

执行要求

- 是否需要调度器: 需要

参数

- `ApInfos`
  - **类型**: `Array`
  - **描述**: 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 命令

```
svc_subscribe Wifi ScanApInfosUpdated
```

SoftApEventHappened

描述 Emitted when a SoftAP event occurs.

执行要求

- 是否需要调度器: 需要

参数

- `Event`
  - **类型**: `String`
  - **描述**: SoftAP event. Allowed values: [Started, Stopped]

Schema JSON

CLI 命令

```
svc_subscribe Wifi SoftApEventHappened
```

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

4.2.4 音频

- 组件注册表：espressif/brookesia_service_audio
- 辅助头文件：`#include "brookesia/service_helper/audio.hpp"`
- 辅助类：`esp_brookesia::service::helper::Audio`

概述

brookesia_service_audio 是为 ESP-Brookesia 生态系统提供的音频服务，提供：

- **音频播放**：支持从 URL 播放音频文件，支持暂停、恢复、停止等播放控制。
- **音频编解码**：支持多种音频编解码格式（PCM、OPUS、G711A），支持编码和解码功能。
- **音量控制**：支持音量设置和查询，提供完整的音量管理能力。
- **播放状态管理**：实时跟踪播放状态（空闲、播放中、暂停），并通过事件通知状态变化。
- **编码器管理**：支持编码器的启动、停止和配置，可设置编码器读取数据大小。
- **解码器管理**：支持解码器的启动、停止和数据输入，支持流式解码。
- **持久化存储**：可选搭配 *brookesia_service_nvs* 服务持久化保存音量等信息。

功能特性

音频编解码格式 Audio Service 支持以下音频编解码格式：

格式	编码	解码	说明
PCM	✓	✓	无损音频格式
OPUS	✓	✓	支持 VBR 和固定比特率配置
G711A	✓	✓	电话音质音频格式

播放控制

- 支持从 URL 播放音频文件。
- 支持暂停、恢复、停止等基础播放控制。
- 提供播放状态事件通知，便于业务层同步 UI 与状态。

编码器配置

- **编解码格式**：PCM、OPUS、G711A。
- **通道数**：1-4 通道。
- **采样位数**：8、16、24、32 位。
- **采样率**：8000、16000、24000、32000、44100、48000 Hz。
- **帧时长**：可配置的帧时长（毫秒）。
- **OPUS 扩展配置**：VBR 开关、比特率设置。

解码器配置

- **编解码格式**：PCM、OPUS、G711A。
- **通道数**：1-4 通道。
- **采样位数**：8、16、24、32 位。
- **采样率**：8000、16000、24000、32000、44100、48000 Hz。
- **帧时长**：可配置的帧时长（毫秒）。

事件通知

- **播放状态变化**：播放状态改变时触发（Idle、Playing、Paused）。
- **编码器事件**：编码器事件发生时触发。
- **编码数据就绪**：编码数据准备好时触发。

服务接口

函数

`SetPlaybackConfig`

描述 Set playback config. Call before starting the service.

执行要求

- 是否需要调度器: 不需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call Audio SetPlaybackConfig {"Config":null}
```

`SetEncoderStaticConfig`

描述 Set encoder static config. Call before starting the service.

执行要求

- 是否需要调度器: 不需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call Audio SetEncoderStaticConfig {"Config":null}
```

`SetDecoderStaticConfig`

描述 Set decoder static config. Call before starting the service.

执行要求

- 是否需要调度器: 不需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call Audio SetDecoderStaticConfig {"Config":null}
```

`SetAFE_Config`

描述 Set AFE config. Call before starting the service.

执行要求

- 是否需要调度器: 不需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call Audio SetAFE_Config {"Config":null}
```

GetAFE_WakeWords

描述 Get AFE wake words. Return type: JSON array<string>. Example: [“ni hao xiao zhi” ,” hello brookesia”]

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio GetAFE_WakeWords
```

PauseAFE_WakeupEnd

描述 Pause AFE wakeup-end task.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio PauseAFE_WakeupEnd
```

ResumeAFE_WakeupEnd

描述 Resume AFE wakeup-end task.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio ResumeAFE_WakeupEnd
```

PlayUrl

描述 Play audio from a URL. Supports loop and interrupt playback.

执行要求

- 是否需要调度器: 需要

参数

- `Url`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Audio URL, for example: “file://spiffs/example.mp3” .
- `Config`
  - **类型**: `Object`
  - **是否必填**: 可选
  - **默认值**: `{"interrupt":true,"delay_ms":0,"loop_count":0,"loop_interval_ms":0,"timeout_ms":0}`
  - **描述**: Playback config. Example: {“interrupt”:true,”delay_ms”:0,”loop_count”:0,”loop_interval_ms”:0,” timeout_ms” :0}

Schema JSON

CLI 命令

```
svc_call Audio PlayUrl {"Url":null,"Config":null}
```

PlayUrls

描述 Play audio from multiple URLs. Supports loop and interrupt playback.

执行要求

- 是否需要调度器: 需要

参数

- `Urls`
  - **类型**: `Array`
  - **是否必填**: 必填
  - **描述**: Audio URL list. Example: [" file://spiffs/example1.mp3" ," file://spiffs/example2.mp3"]
- `Config`
  - **类型**: `Object`
  - **是否必填**: 可选
  - **默 认 值**: `{"interrupt":true,"delay_ms":0,"loop_count":0,"loop_interval_ms":0,"timeout_ms":0}`
  - **描述**: Playback config. Example: {"interrupt":true,"delay_ms":0,"loop_count":0,"loop_interval_ms":0," timeout_ms" :0}

Schema JSON

CLI 命令

```
svc_call Audio PlayUrls {"Urls":null,"Config":null}
```

PlayControl

描述 Control audio playback.

执行要求

- 是否需要调度器: 需要

参数

- `Action`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Playback action. Allowed values: [Pause, Resume, Stop]

Schema JSON

CLI 命令

```
svc_call Audio PlayControl {"Action":null}
```

SetVolume

描述 Set playback volume.

执行要求

- 是否需要调度器: 需要

参数

- `Volume`
  - **类型**: `Number`
  - **是否必填**: 必填
  - **描述**: Volume in range [0, 100].

Schema JSON

CLI 命令

```
svc_call Audio SetVolume {"Volume":null}
```

`GetVolume`

描述 Get playback volume. Return type: number. Example: 70

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio GetVolume
```

`SetMute`

描述 Set playback mute.

执行要求

- 是否需要调度器: 需要

参数

- `Enable`
  - **类型**: `Boolean`
  - **是否必填**: 必填
  - **描述**: Enable mute.

Schema JSON

CLI 命令

```
svc_call Audio SetMute {"Enable":null}
```

StartEncoder

描述 Start audio encoder.

执行要求

- 是否需要调度器: 需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call Audio StartEncoder {"Config":null}
```

StopEncoder

描述 Stop audio encoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio StopEncoder
```

PauseEncoder

描述 Pause audio encoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio PauseEncoder
```

ResumeEncoder

描述 Resume audio encoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio ResumeEncoder
```

StartDecoder

描述 Start audio decoder.

执行要求

- 是否需要调度器: 需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: Audio decoder config. Example: { “type” :” PCM” ,” general” :{ “channels” :0,” sample_bits” :0,” sample_rate” :0,” frame_duration” :0}}

Schema JSON

CLI 命令

```
svc_call Audio StartDecoder {"Config":null}
```

StopDecoder

描述 Stop audio decoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio StopDecoder
```

FeedDecoderData

描述 Feed audio data to the decoder.

执行要求

- 是否需要调度器: 不需要

参数

- Data
  - **类型**: RawBuffer
  - **是否必填**: 必填
  - **描述**: Audio data to decode.

Schema JSON

CLI 命令

```
svc_call Audio FeedDecoderData {"Data":null}
```

ResetData

描述 Reset audio data. Includes player volume.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Audio ResetData
```

事件

PlayStateChanged

描述 Emitted when playback state changes.

执行要求

- 是否需要调度器: 需要

参数

- `State`
  - **类型**: `String`
  - **描述**: Playback state. Allowed values: [Idle, Playing, Paused]

Schema JSON

CLI 命令

```
svc_subscribe Audio PlayStateChanged
```

AFE_EventHappened

描述 Emitted when an AFE event occurs.

执行要求

- 是否需要调度器: 需要

参数

- `Event`
  - **类型**: `String`
  - **描述**: AFE event. Allowed values: [VAD_Start, VAD_End, WakeStart, WakeEnd]

Schema JSON

CLI 命令

```
svc_subscribe Audio AFE_EventHappened
```

EncoderDataReady

描述 Emitted when encoder data is ready.

执行要求

- 是否需要调度器: 不需要

参数

- `Data`
  - **类型**: `RawBuffer`
  - **描述**: Encoded audio data.

Schema JSON

CLI 命令

```
svc_subscribe Audio EncoderDataReady
```

`RecorderDataReady`

描述 Emitted when recorder data is ready.

执行要求

- 是否需要调度器: 不需要

参数

- `Data`
  - **类型**: `RawBuffer`
  - **描述**: Recorded raw audio data.

Schema JSON

CLI 命令

```
svc_subscribe Audio RecorderDataReady
```

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

4.2.5 视频

- 组件注册表：espressif/brookesia_service_video
- 辅助头文件：`#include "brookesia/service_helper/video.hpp"`
- 辅助类：`esp_brookesia::service::helper::Video`

概述

brookesia_service_video 是为 ESP-Brookesia 生态系统提供的视频服务，主要包括：

- **视频编码**：将摄像头等采集源编码为多种压缩或原始格式（分辨率、帧率、格式可分别配置）。
- **视频解码**：将 H.264、MJPEG 等压缩码流解码为常用 RGB、YUV 等显示或后处理格式。

功能特性

编码器 视频编码服务的输入来自 ** 本地视频设备 **，默认设备路径为 `/dev/video0`，可在 Kconfig 中配置默认路径前缀与数量。

视频编码服务可将一路输出配置为下列类型之一（每路可单独指定分辨率与帧率）：

类型	说明
H.264	常用网络与存储压缩格式
MJPEG	逐帧 JPEG 类压缩
RGB565 / RGB888 / BGR888	RGB 类格式
YUV420 / YUV422 / O_UYY_E_VYY	YUV 类格式

> **警告：** 目前暂不支持多路输出。

解码器 解码服务的输入为压缩码流格式，输出为像素格式，可按业务分别配置。常见取值如下：

输入（压缩格式）

格式	说明
H.264	常用网络与存储视频压缩格式
MJPEG	逐帧 JPEG 类压缩码流

输出（像素格式）

格式	说明
RGB565（大端 / 小端）	16 位 RGB，适合部分屏驱与显存布局
RGB888 / BGR888	24 位 RGB / BGR，常用于显示与图像处理
YUV420P / YUV422P	平面 YUV，便于视频处理管线
YUV422 / UYVY422	打包 YUV，常见于采集与显示链路
O_UYY_E_VYY	特定打包 YUV 布局（视硬件与管线而定）

服务接口

函数

编码器

`Open`

描述 Open the encoder with config.

执行要求

- 是否需要调度器: 需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call VideoEncoder0 Open {"Config":null}
```

`Close`

描述 Close encoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call VideoEncoder0 Close
```

`Start`

描述 Start encoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call VideoEncoder0 Start
```

Stop

描述 Stop encoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call VideoEncoder0 Stop
```

FetchFrame

描述 Fetch an encoder output frame and emit *FetchSinkFrameReady*. Only available in non-stream mode.

执行要求

- 是否需要调度器: 需要

参数

- `SinkIndex`
  - **类型**: `Number`
  - **是否必填**: 可选
  - **默认值**: `0E0`
  - **描述**: Sink index.

Schema JSON

CLI 命令

```
svc_call VideoEncoder0 FetchFrame {"SinkIndex":null}
```

解码器

Open

描述 Open the decoder with config.

执行要求

- 是否需要调度器: 需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call VideoDecoder0 Open {"Config":null}
```

`Close`

描述 Close decoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call VideoDecoder0 Close
```

`Start`

描述 Start decoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call VideoDecoder0 Start
```

Stop

描述 Stop decoder.

执行要求

- 是否需要调度器: 需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call VideoDecoder0 Stop
```

FeedFrame

描述 Feed a decoder input frame.

执行要求

- 是否需要调度器: 需要

参数

- Frame
  - **类型**: RawBuffer
  - **是否必填**: 必填
  - **描述**: Frame data.

Schema JSON

CLI 命令

```
svc_call VideoDecoder0 FeedFrame {"Frame":null}
```

事件

编码器

StreamSinkFrameReady

描述 Emitted when an encoder stream frame is ready. Stream mode only.

执行要求

- 是否需要调度器: 不需要

参数

- `SinkIndex`
  - **类型**: `Number`
  - **描述**: Sink index.
- `SinkInfo`
  - **类型**: `Object`
  - **描述**: Sink info. Example: { “format” :” H264” ,” width” :320,” height” :240,” fps” :30}
- `Frame`
  - **类型**: `RawBuffer`
  - **描述**: Encoded frame data.

Schema JSON

CLI 命令

```
svc_subscribe VideoEncoder0 StreamSinkFrameReady
```

FetchSinkFrameReady

描述 Emitted when an encoder fetched frame is ready. Non-stream mode only.

执行要求

- 是否需要调度器: 不需要

参数

- `SinkIndex`
  - **类型**: `Number`
  - **描述**: Sink index.
- `SinkInfo`
  - **类型**: `Object`
  - **描述**: Sink info. Example: { “format” :” MJPEG” ,” width” :320,” height” :240,” fps” :15}
- `Frame`
  - **类型**: `RawBuffer`
  - **描述**: Encoded frame data.

Schema JSON

CLI 命令

```
svc_subscribe VideoEncoder0 FetchSinkFrameReady
```

解码器

SinkFrameReady

描述 Emitted when a decoder output frame is ready.

执行要求

- 是否需要调度器: 不需要

参数

- `Width`
  - **类型**: `Number`
  - **描述**: Decoded frame width.
- `Height`
  - **类型**: `Number`
  - **描述**: Decoded frame height.
- `Frame`
  - **类型**: `RawBuffer`
  - **描述**: Decoded frame data.

Schema JSON

CLI 命令

```
svc_subscribe VideoDecoder0 SinkFrameReady
```

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

4.2.6 服务自定义

- 组件注册表：espressif/brookesia_service_custom
- 公共头文件：`#include "brookesia/service_custom.hpp"`

概述

brookesia_service_custom 为 ESP-Brookesia 生态系统提供即用的 **CustomService**，支持用户自定义 function 和 event 的注册与调用，无需创建独立的服务组件。

典型使用场景

- **轻量级功能**：对于无需封装为完整 Brookesia 组件的功能（如 LED 控制、PWM、GPIO 翻转、简单传感器等），可通过 CustomService 的接口进行封装和调用。
- **快速原型**：将应用逻辑快速暴露为可调用的 function 或 event，支持本地调用或远程 RPC 访问。
- **可扩展性**：在不修改服务框架的前提下，为应用添加自定义能力。

功能特性

- **动态注册**：运行时通过 `register_function()` 和 `register_event()` 注册 function 和 event。
- **固定 Handler 签名**：Handler 接收 `FunctionParameterMap`，返回 `FunctionResult`；支持 lambda、`std::function`、自由函数、仿函数、`std::bind`。
- **事件发布/订阅**：完整的事件生命周期：注册、发布、订阅、注销。
- **ServiceManager 集成**：与 ServiceManager 配合，支持本地调用和远程 RPC。
- **可选 Worker**：可配置任务调度器，实现线程安全执行。

API 参考

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.

参数
- **schema** –**[in]** Function schema
- **handler** –**[in]** Function handler

返回 true if registered successfully, false otherwise

bool **unregister_function**(const std::string &function_name)

Unregister a function.

参数 **function_name** –**[in]** Function name
返回 true if unregistered successfully, false otherwise

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

Get currently registered function schemas for custom service.

返回 std::vector<FunctionSchema> Function schemas

bool **register_event**(EventSchema event_schema)

Register an event for custom service.

参数 **event_schema** –**[in]** Event schema to register.
返回 true if registered successfully, false otherwise

bool **unregister_event**(const std::string &event_name)

Unregister an event.

参数 **event_name** –**[in]** Event name
返回 true if unregistered successfully, false otherwise

bool **publish_event**(const std::string &event_name, EventItemMap event_items)

Publish an event for custom service.

参数
- **event_name** –**[in]** Event name
- **event_items** –**[in]** Event items

返回 true if published successfully, false otherwise

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

Get currently registered event schemas for custom service.

返回 std::vector<EventSchema> Event schemas

Public Static Functions

static inline *CustomService* &**get_instance**()

Get the process-wide singleton instance.

返回 Reference to the singleton custom service.

4.3 开发指南

4.3.1 使用说明

本文说明如何在 ESP-IDF 工程中集成 ESP-Brookesia **服务框架**，并以 **Wi-Fi 服务**为例给出典型调用方式；完整可编译示例见 `examples/service/wifi`。

在常规工程中，直接使用某个组件时，一般需要添加依赖、包含头文件并调用其接口。功能增多后，模块与组件之间的耦合会变强；若要移除某一组件，除删除依赖外，还需清理相关头文件与调用，维护成本较高。

ESP-Brookesia 将上述关系抽象为统一的 **函数**与 **事件**两类接口。多数场景下只需依赖 `brookesia_service_manager` 与 `brookesia_service_helper`，即可通过 Helper 访问各服务功能；也可在运行时检查某服务是否可用，未链接具体服务组件时通常不会因 Helper 调用本身而产生编译错误，便于裁剪与维护。

此外，框架借助任务调度器 等机制支持服务侧异步执行，减轻对调用线程的阻塞；对外暴露的接口在设计上考虑线程安全，降低业务侧自行处理并发同步的负担。

下文按 **依赖配置 → 初始化与绑定 → 调用函数 → 订阅事件 → 事件监听器**的顺序说明操作要点。

添加组件依赖

在工程根目录或组件目录的 `idf_component.yml` 中声明依赖。以 Wi-Fi 服务为例：

```
dependencies:
  espressif/brookesia_service_wifi: "*"
  # espressif/brookesia_service_nvs: "*" # 可选，如果需要使用 NVS 服务
```

若工程不链接具体服务实现，仅需满足「能编译通过 Helper 相关代码」一类约束时，可只添加 `brookesia_service_helper`：

```
dependencies:
  espressif/brookesia_service_helper: "*"
```

组件的获取与版本约束请参考如何获取和使用组件。

头文件、命名空间与类型别名

```
#include "brookesia/lib_utils.hpp"
#include "brookesia/service_manager.hpp"
// Helper 头文件，需要根据具体服务组件选择对应的 Helper 头文件
#include "brookesia/service_helper/wifi.hpp"
```

(下页继续)

(续上页)

```
// 所有 Brookesia 组件中的数据类型均在 esp_brookesia 命名空间下
using namespace esp_brookesia;
// 使用类型别名简化代码
// 服务相关的类型则位于 service 命名空间下
using WifiHelper = service::helper::Wifi;
```

启动服务管理器

在调用任何服务接口前，应对全局 **单例** `ServiceManager` 调用 `start()` 完成初始化。

```
auto &service_manager = service::ServiceManager::get_instance();
service_manager.start();
```

启动/停止服务

```
// 在绑定前可检查 Helper 对应的服务是否已链接到工程
if (!WifiHelper::is_available()) {
    // 服务不可用：未添加对应服务组件时不会因此产生编译错误，但此处检查会失败
    return;
}

// 获取服务绑定，绑定期间会拉起该服务及其依赖；
// 在 `binding` 对象存活期间服务保持运行，离开作用域后解绑（析构），服务停止。
// 如果需要长期持有服务，可以将 `binding` 对象存储到非栈区域中。
auto binding = service_manager.bind(WifiHelper::get_name().data());
if (!binding.is_valid()) {
    // 服务启动失败
    return;
}
```

调用服务函数

调用前请确认目标函数的 **参数类型、顺序与返回值**，可在对应 Helper 的契约文档或头文件中查阅，例如：

- *Wi-Fi 服务函数接口说明文档*
- Wi-Fi 服务辅助头文件源码

备注:

- **描述**：包含函数的功能描述、返回值信息等。若无 **返回值**相关内容，则表示函数无返回值。
- **参数列表：包含函数参数的名称、类型、描述和默认值。**
  - **类型：总共支持 `String`、`Number`、`Boolean`、`Object`、`Array`、`RawBuffer` 六种，它们与实际调用**
    - `String`: `std::string`
    - `Number`: `double/int/<` 任意算术类型`>`
    - `Boolean`: `bool`
    - `Object`: `boost::json::object`（通常可以利用 `struct` 序列化生成）
    - `Array`: `boost::json::array`（通常可以利用 `std::vector` 序列化生成）
    - `RawBuffer`: `service::RawBuffer`
  - **描述：**
    - 当参数类型为 `String` 且描述中提供候选值时，表示参数可以通过序列化将枚举类型传入。
    - 当参数类型为 `Object/Array` 时，描述会提供示例。
  - **默认值**：当参数标记为 **可选**时，会提供默认值。

- **执行要求：表示该函数是否需要调度器（Task Scheduler）支持。**
  - 若标记为 **需要**，则表示该函数必须在 **服务启动后**再执行，此时支持通过调度器进行 **同步/异步调用**。
  - **若标记为 不需要，则表示该函数可以在 服务启动前执行，此时仅支持 同步调用。这种情况通常出现在以**

    * 该函数需要在服务启动前执行。
    * 该函数的参数列表中存在 `RawBuffer` 类型。
    * 该函数的底层实现已经通过锁或其他机制保证了线程安全，因此无需通过调度器进行调用。

通过 Helper 的静态方法可发起 **同步调用**或 **异步调用**。先说明 **同步调用**：

同步调用 此接口会阻塞等待函数执行完成，并返回结果；若超时则返回错误。

```
auto result = WifiHelper::call_function_sync<返回类型>
↪(WifiHelper::FunctionId::函数名 [, 参数列表...] [, 超时时间]);
if (!result) {
   // 调用失败，打印错误信息
   BROOKESIA_LOGE("Failed: %1%", result.error());
} else {
   // 调用成功
   // 若函数有返回值，则可以通过 `result.value()` 获取
   auto &value = result.value(); // `value` 类型为 `返回类型`
   // ...
}
```

备注:

- 参数列表的类型、数量和顺序必须与函数定义中的参数列表一致，若函数无参数，则省略。
- 若 返回类型为 `void`，则 < 返回类型> 模板参数可以省略。
- 超时时间为可选参数,若省略则使用默认值`BROOKESIA_SERVICE_MANAGER_DEFAULT_CALL_FUNCTION_TIMEO`
- 具体接口说明见*esp_brookesia::service::helper::Base::call_function_sync()*。

示例：多参数输入 以 `SetConnectAp` 为例，接口要点与示例代码如下：

- 函数名：`SetConnectAp`
- **参数列表：**
  - `SSID`：类型为 `String`
  - `Password`：类型为 `String`
- 返回值类型：`void`

```
// 严格按照参数列表的顺序传入参数，不可使用默认值
auto set_connect_ap_result = WifiHelper::call_function_sync(
   WifiHelper::FunctionId::SetConnectAp, "ssid1", "password1",␣
↪service::helper::Timeout(100));
if (!set_connect_ap_result) {
   // 调用失败，打印错误信息
   BROOKESIA_LOGE("Failed: %1%", set_connect_ap_result.error());
}
```

示例：序列化生成参数 以 `TriggerGeneralAction` 为例，接口要点与示例代码如下：

- 函数名：`TriggerGeneralAction`
- **参数列表：**
  - `Action`：类型为 `String` （可由 `WifiHelper::GeneralAction` 类型数据序列化生成）

- 返回值类型：void

```
auto start_result = WifiHelper::call_function_sync(
    WifiHelper::FunctionId::TriggerGeneralAction,
    BROOKESIA_DESCRIBE_TO_STR(WifiHelper::GeneralAction::Start));
if (!start_result) {
    // 调用失败，打印错误信息
    BROOKESIA_LOGE("Failed: %1%", start_result.error());
}
```

以 SetScanParams 为例，接口要点与示例代码如下：

- 函数名：SetScanParams
- **参数列表：**
  - Param：类型为 Object （可由 WifiHelper::ScanParams 类型数据序列化生成）
- 返回值类型：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) {
    // 调用失败，打印错误信息
    BROOKESIA_LOGE("Failed: %1%", set_scan_params_result.error());
}
```

示例：返回值解析 以 GetConnectedAps 为例，接口要点与示例代码如下：

- 函数名：GetConnectedAps
- 参数列表：无
- 返回值类型：boost::json::array （可被反序列化为 std::vector<WifiHelper::ConnectApInfo> 类型数据）

```
auto get_connected_aps_result =
    WifiHelper::call_function_sync<boost::json::array>
↪(WifiHelper::FunctionId::GetConnectedAps);
if (!get_connected_aps_result) {
    // 调用失败，打印错误信息
    BROOKESIA_LOGE("Failed to get connected APs: %1%", get_connected_aps_result.
↪error());
}

// 解析返回值
std::vector<WifiHelper::ConnectApInfo> infos;
auto parse_result = BROOKESIA_DESCRIBE_FROM_JSON(get_connected_aps_result.value(),␣
↪infos);
if (!parse_result) {
    // 解析失败，打印错误信息
    BROOKESIA_LOGE("Failed to parse connected APs: %1%", get_connected_aps_result.
↪error());
} else {
    // 解析成功，打印结果
    BROOKESIA_LOGI("Connected APs: %1%", infos);
}
```

异步调用 异步调用在 **提交**调用后立即返回，不阻塞当前线程；若传入结果回调，则在函数执行完成后由框架在合适时机 **异步**回调。

```
auto on_function_handler = [](service::FunctionResult &&result) {
    // 处理返回值
    if (!result.success) {
        // 执行失败，打印错误信息
        BROOKESIA_LOGE("Failed: %1%", result.error_message);
    } else {
        // 执行成功
        // 若函数有返回值
        // auto &value = result.get_data<返回类型>();
        // ...
    }
};
auto result = WifiHelper::call_function_async(WifiHelper::FunctionId::函数名 [,
↪参数列表...] [, on_function_handler]);
if (!result) {
    // 调用失败
}
```

备注:

- 参数列表的类型、数量和顺序必须与函数定义中的参数列表一致，若函数无参数，则省略。
- on_function_handler 为可选参数，若省略则不处理返回值。
- 如果程序对同一个服务串行执行多个异步调用，则这些调用的实际执行顺序与调用顺序一致。如：

```
Helper::call_function_async(Helper::FunctionId::Function1);
Helper::call_function_async(Helper::FunctionId::Function2);
Helper::call_function_async(Helper::FunctionId::Function3);
// 服务内部实际执行顺序为：Function1 -> Function2 -> Function3
```

- 具体接口说明见*esp_brookesia::service::helper::Base::call_function_async()*。

示例 以 GetConnectedAps 为例，接口要点与示例代码如下：

- 函数名：GetConnectedAps
- 参数列表：无
- 返回值类型：boost::json::array（可由 std::vector<WifiHelper::ConnectApInfo> 类型数据序列化生成）

```
auto on_get_connected_aps_handler = [](service::FunctionResult &&result) {
    if (!result.success) {
        // 执行失败，打印错误信息
        BROOKESIA_LOGE("Failed to get connected APs: %1%", result.error_message);
        return;
    }
    // 执行成功，解析返回值
    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) {
        // 解析失败，打印错误信息
        BROOKESIA_LOGE("Failed to parse connected APs: %1%", result);
        return;
    }
    // 解析成功，打印结果
    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) {
```

(下页继续)

(续上页)

```
    // 调用失败
    return;
}
```

订阅服务事件

订阅前请确认事件的 **条目名称、类型与顺序**，可在 Helper 文档或头文件中查阅，例如：

- *Wi-Fi 服务事件接口说明文档*
- Wi-Fi 服务辅助头文件源码

备注：

- **描述**：包含事件的功能描述、返回值信息等。若无 **返回值**相关内容，则表示事件无返回值。
- **条目列表：包含事件条目的名称、类型、描述和默认值。**
  - **类型：总共支持 `String`、`Number`、`Boolean`、`Object`、`Array`、`RawBuffer` 六种，它们与实际调用**
    - `String`: `std::string`
    - `Number`: `double/int/< 任意算术类型>`
    - `Boolean`: `bool`
    - `Object`: `boost::json::object`` （通常可以利用 ``struct` 序列化生成）
    - `Array`: `boost::json::array`` （通常可以利用 ``std::vector` 序列化生成）
    - `RawBuffer`: `service::RawBuffer`
  - **描述：**
    - 当条目类型为 `String` 且描述中提供候选值时，表示条目可以被反序列化为枚举类型数据。
    - 当条目类型为 `Object/Array` 时，描述会提供示例。
- **执行要求：表示该事件是否需要调度器（Task Scheduler）支持。**
  - 若标记为 **需要**，则表示该事件必须在 **服务启动后**再发布。
  - **若标记为 不需要，则表示该事件可以在 服务启动前发布。这种情况通常出现在以下场景：**
    - 该事件需要在服务启动前执行。
    - 该事件的条目列表中存在 `RawBuffer` 类型。

订阅事件 使用 `subscribe_event()` 注册回调，在事件触发时按订阅顺序串行调用：

```
auto on_event_handler = [](const std::string &event_name[, 条目列表...]) {
    // 处理事件
};
// 返回值 `conn` 表示订阅连接；析构时会 **自动取消订阅**（RAII）。
// 若需长期保持订阅，请将连接对象保存在非栈上（如静态或堆上）。
auto conn = WifiHelper::subscribe_event(WifiHelper::EventId::事件名, on_event_
↪handler);
if (!conn.connected()) {
    // 订阅失败
}
// 除此之外，也可以手动取消订阅
conn.disconnect();
```

备注：

- `subscribe_event()` 须在 `ServiceManager::start()` 之后调用。
- 条目列表的类型、数量和顺序必须与事件定义中的条目列表一致；若事件无条目，则省略。

- 服务事件的订阅数量没有代码限制，某一事件触发时，所有订阅该事件的回调函数都会按照订阅顺序被串行调用。
- 具体接口说明见*esp_brookesia::service::helper::Base::subscribe_event()*。

示例 以 GeneralEventHappened 为例，接口要点与示例代码如下：

- 事件名：GeneralEventHappened
- **条目列表：**
  - Event：类型为 String``（可被反序列化为 ``WifiHelper::GeneralEvent 类型数据）
  - IsUnexpected：类型为 Boolean

```
auto on_general_event_happened_handler = [](const std::string &event_name, const␣
↪std::string &event,
                                            bool is_unexpected) {
    // 事件触发，解析条目
    WifiHelper::GeneralEvent general_event;
    auto parse_result = BROOKESIA_DESCRIBE_STR_TO_ENUM(event, general_event);
    if (!parse_result) {
        // 解析失败，打印错误信息
        BROOKESIA_LOGE("Failed to parse general event: %1%", event);
        return;
    }
    // 解析成功，打印结果
    BROOKESIA_LOGI("General event: %1%", general_event);
};
// 连接析构时自动取消订阅（RAII）
auto general_event_happened_connection = WifiHelper::subscribe_event(
    WifiHelper::EventId::GeneralEventHappened, on_general_event_happened_handler);
if (!general_event_happened_connection.connected()) {
    // 订阅失败
    return;
}

// 触发 Start 动作
WifiHelper::call_function_async(WifiHelper::FunctionId::TriggerGeneralAction,
                                BROOKESIA_DESCRIBE_TO_
↪STR(WifiHelper::GeneralAction::Start));

// 等待事件触发（示例用延时，实际工程请用同步原语等）
boost::this_thread::sleep_for(boost::chrono::seconds(5));
```

以 ScanApInfosUpdated 为例，接口要点与示例代码如下：

- 事件名：ScanApInfosUpdated
- **条目列表：**
  - ApInfos:类型为Array``(可被反序列化为 ``std::vector<WifiHelper::ScanApInfo>类型数据)

```
auto on_scan_ap_infos_updated_handler = [](const std::string &event_name, const␣
↪boost::json::array &ap_infos) {
    // 事件触发，解析条目
    std::vector<WifiHelper::ScanApInfo> scanned_aps;
    auto parse_result = BROOKESIA_DESCRIBE_FROM_JSON(ap_infos, scanned_aps);
    if (!parse_result) {
        // 解析失败，打印错误信息
        BROOKESIA_LOGE("Failed to parse scan AP infos: %1%", ap_infos);
        return;
    }
    // 解析成功，打印结果
```

(下页继续)

(续上页)

```
    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()) {
    // 订阅失败
    return;
}

WifiHelper::call_function_async(WifiHelper::FunctionId::TriggerScanStart);

boost::this_thread::sleep_for(boost::chrono::seconds(10));
```

事件监听器 EventMonitor 用于在 **异步事件**之上提供 **同步等待**语义：可在调用某接口后阻塞等待指定事件或任意事件，便于编排流程。

```
// 创建并启动监听器
WifiHelper::EventMonitor<WifiHelper::EventId::事件名> event_monitor;
event_monitor.start();

// 调用会触发该事件的接口，略

// 等待任意一次事件
auto got_any_event = event_monitor.wait_for_any(超时时间);
if (!got_any_event) {
    // 超时，事件未触发
    return;
}

// 或等待条目匹配的事件
auto got_event = event_monitor.wait_for(std::vector<service::EventItem>{[条目列表..
↪.]}, 超时时间);
if (!got_event) {
    // 超时，未匹配到指定条目
    return;
}

event_monitor.stop();
```

备注:

- 条目列表的类型、数量和顺序必须与事件定义中的条目列表一致。
- 具体接口说明见 esp_brookesia::service::helper::EventMonitor。

示例：等待特定条目列表的事件触发 以 GeneralEventHappened 为例，监听器用法如下：

- 事件名：GeneralEventHappened
- **条目列表:**
  - Event：类型为 String`` （可由 ``WifiHelper::GeneralEvent 类型数据序列化生成）
  - IsUnexpected：类型为 Boolean

```
WifiHelper::EventMonitor<WifiHelper::EventId::GeneralEventHappened> general_event_
↪monitor;
BROOKESIA_CHECK_FALSE_EXIT(general_event_monitor.start(), "Failed to start general␣
↪event monitor");
```

(下页继续)

(续上页)

```
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) {
    // 超时，事件未触发
    return;
}
```

示例：获取最新接收的事件条目列表 以 ScanApInfosUpdated 为例，监听器用法如下：

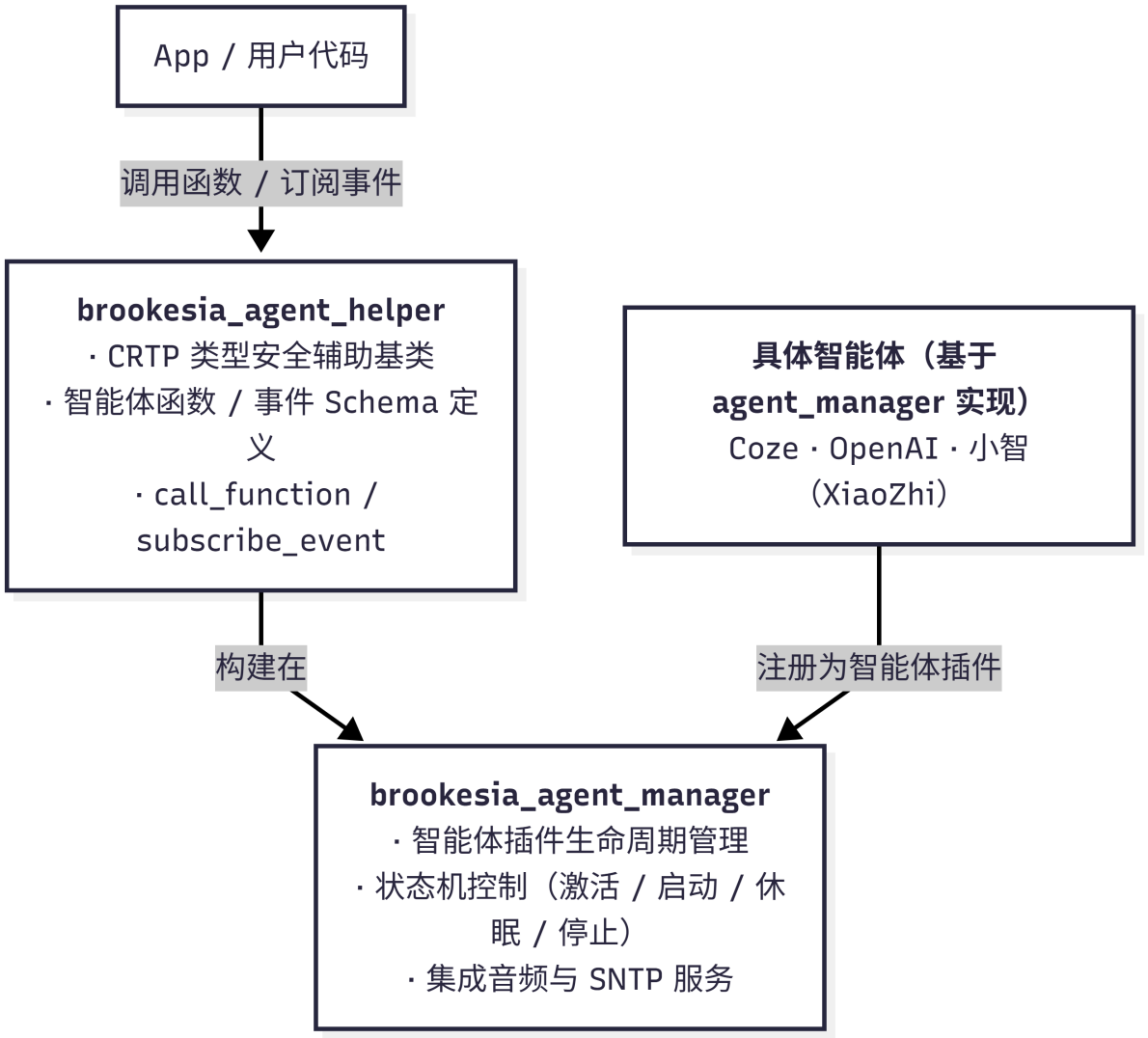
- 事件名：ScanApInfosUpdated
- **条目列表：**
  - ApInfos:类型为Array``(可被反序列化为 ``std::vector<WifiHelper::ScanApInfo> 类型数据)

```
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) {
    // 超时，事件未触发
    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

AI 智能体组件

本分类包含 ESP-Brookesia 智能体框架组件的说明内容。ESP-Brookesia 智能体框架由智能体框架层和具体智能体层组成，各组件的层级关系如下：



- `brookesia_agent_manager`：智能体框架核心，负责智能体插件注册、状态机生命周期控制，并集成音频与时间同步服务

- `brookesia_agent_helper`：基于 CRTP 的类型安全辅助层，简化智能体的函数/事件定义与调用方式，构建在 `service_helper` 之上
- **具体智能体**：基于 `agent_manager` 实现的特定 AI 平台接入，注册到框架后可被上层统一管理和调用

5.1 智能体框架

5.1.1 AI 智能体管理器

- 组件注册表：espressif/brookesia_agent_manager
- 辅助头文件：`#include "brookesia/agent_helper/manager.hpp"`
- 辅助类：`esp_brookesia::agent::helper::Manager`

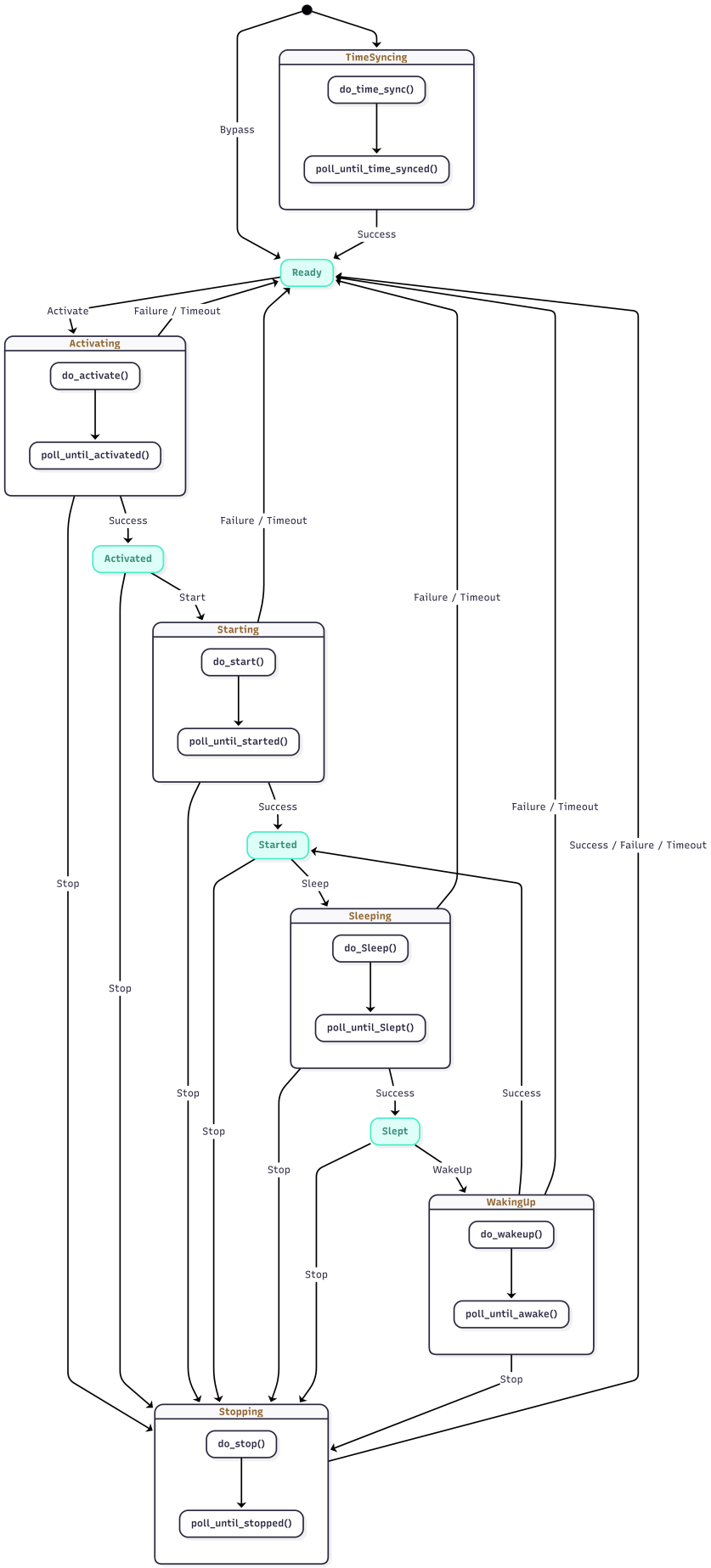
概述

brookesia_agent_manager 是 ESP-Brookesia 智能体管理框架核心组件，提供：

- **统一的智能体生命周期管理**：通过插件机制，集中式管理智能体的初始化、激活、启动、停止、休眠和唤醒，支持智能体之间的动态切换。
- **状态机管理**：基于状态机自动管理智能体的状态转换，确保状态转换的正确性和一致性。
- **智能体操作控制**：支持智能体的暂停/恢复、中断说话、状态查询等操作，提供完整的智能体控制能力。
- **对话模式支持**：支持实时对话（RealTime）和手动对话（Manual）两种模式，手动模式下支持手动开始/停止监听。
- **事件驱动架构**：支持通用操作事件、状态变化事件、说话/监听状态事件、文本交互事件和表情事件，实现智能体与应用之间的解耦通信。
- **功能扩展支持**：支持函数调用、文本处理、中断说话、表情等扩展功能，通过智能体属性配置灵活启用。
- **AFE 事件处理**：可选启用 AFE（音频前端）事件处理，自动响应唤醒开始/结束事件。
- **服务集成**：基于服务框架，提供统一的服务接口，集成音频服务和 SNTP 服务。
- **持久化存储**：可选搭配 *brookesia_service_nvs* 服务持久化保存智能体激活状态和对话模式等信息。

状态机架构

brookesia_agent_manager 使用状态机管理智能体的生命周期，确保状态转换的正确性和一致性。


TimeSyncing
do_time_sync()
poll_until_time_synced()
Bypass
Success
Ready
Activate
Failure / Timeout
Activating
do_activate()
poll_until_activated()
Success
Failure / Timeout
Activated
Start
Failure / Timeout
Starting
do_start()
poll_until_started()
Success
Failure / Timeout
Started
Success / Failure / Timeout
Stop
Sleep
Sleeping
do_Sleep()
poll_until_Slept()
Stop
Stop
Success
Success
Stop
Slept
Stop
WakeUp
WakingUp
do_wakeup()
poll_until_awake()
Stop
Stop
Stopping
do_stop()
poll_until_stopped()

状态机图

状态说明

状态	类型	说明
TimeSyncing	瞬态	正在同步时间，等待时间同步完成事件
Ready	稳定	就绪状态，时间已同步（或跳过），等待激活命令
Activating	瞬态	正在激活智能体，等待激活完成事件
Activated	稳定	智能体已激活，等待启动命令
Starting	瞬态	正在启动智能体，等待启动完成事件
Started	稳定	智能体已启动，可以接收音频输入和输出
Sleeping	瞬态	正在休眠智能体，等待休眠完成事件
Slept	稳定	智能体已休眠，可以唤醒或停止
WakingUp	瞬态	正在唤醒智能体，等待唤醒完成事件
Stopping	瞬态	正在停止智能体，等待停止完成事件

API 参考

智能体基类 公共头文件：`#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.

返回 const AgentAttributes& Agent attributes passed at construction time.

inline const AudioConfig &**`get_audio_config`**() const

Get the audio configuration used by the agent.

返回 const AudioConfig& Current encoder and decoder configuration.

智能体管理器 公共头文件：`#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.

返回 *Manager*& Singleton instance.

5.1.2 AI 智能体辅助

- 组件注册表：espressif/brookesia_agent_helper
- 公共头文件：`#include "brookesia/agent_helper.hpp"`

概述

brookesia_agent_helper 是 ESP-Brookesia 智能体开发辅助库，基于 C++20 Concepts 和 CRTP（Curiously Recurring Template Pattern）模式，为智能体开发者和使用者提供类型安全的定义、Schema 和统一调用接口。

特性

- 类型安全定义：提供强类型枚举和结构体类型定义，确保编译时类型检查。
- Schema 定义：提供标准化的函数与事件 Schema，覆盖名称、参数类型与描述等元数据。
- 统一调用接口：提供类型安全的同步与异步函数调用接口，自动处理类型转换与错误处理。
- 事件订阅接口：提供类型安全的事件订阅接口，支持事件处理回调。

模块介绍

管理器 Helper

- 公共头文件：`#include "brookesia/agent_helper/manager.hpp"`

概述 本页用于查看 Manager helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 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.

返回 std::string Agent name.

inline bool **is_general_functions_supported**(*AgentGeneralFunction* function) const

Check whether an optional general function is supported.

参数 **function** –**[in]** Function to check.

返回 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.

参数 **event** –**[in]** Event to check.

返回 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

- 公共头文件：`#include "brookesia/agent_helper/coze.hpp"`

概述 本页用于查看 Coze helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 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.

返回 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.

返回 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

- 公共头文件：`#include "brookesia/agent_helper/openai.hpp"`

概述 本页用于查看 OpenAI helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 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.

返回 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.

返回 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.

小智 Helper

- 公共头文件：`#include "brookesia/agent_helper/xiaozhi.hpp"`

概述 本页用于查看小智 helper 的原始 Doxygen API，包括公共类型、枚举、方法与相关宏定义。

API 参考

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.

返回 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.

返回 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.

返回 std::span<const service::EventSchema> Static schema span.

5.2 智能体

5.2.1 Coze

- 组件注册表：espressif/brookesia_agent_coze
- 辅助头文件：`#include "brookesia/agent_helper/coze.hpp"`
- 辅助类：`esp_brookesia::agent::helper::Coze`

概述

brookesia_agent_coze 是基于 ESP-Brookesia Agent Manager 框架实现的 Coze 智能体。

功能特性

- **Coze API 集成**：通过 WebSocket 与 Coze 平台进行实时通信，支持语音对话和文本交互。
- **OAuth2 认证**：支持 Coze 平台的 OAuth2 认证机制，自动获取和管理访问令牌。
- **多机器人支持**：支持配置多个机器人，可以动态切换当前激活的机器人。
- **音频编解码**：内置音频编解码支持，默认使用 G711A 格式，支持 16kHz 采样率。
- **表情支持**：支持接收和显示 Coze 平台的表情事件。
- **事件处理**：支持 Coze 平台事件（如余额不足等）的监听和处理。
- **持久化存储**：可选搭配 *brookesia_service_nvs* 服务持久化保存鉴权信息和机器人等信息。

服务接口

函数

`SetActiveRobotIndex`

描述 Set active robot index.

执行要求

- 是否需要调度器: 不需要

参数

- `Index`
  - **类型**: `Number`
  - **是否必填**: 必填
  - **描述**: Robot index to activate.

Schema JSON

CLI 命令

```
svc_call AgentCoze SetActiveRobotIndex {"Index":null}
```

GetActiveRobotIndex

描述 Get active robot index. Return type: number. Example: 0

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call AgentCoze GetActiveRobotIndex
```

GetRobotInfos

描述 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” }]

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call AgentCoze GetRobotInfos
```

事件

CozeEventHappened

描述 Emitted when a Coze event occurs.

执行要求

- 是否需要调度器: 需要

参数

- `CozeEvent`
  - **类型**: `String`
  - **描述**: Coze event. Allowed values: [InsufficientCreditsBalance]

Schema JSON

CLI 命令

```
svc_subscribe AgentCoze CozeEventHappened
```

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

5.2.2 OpenAI

- 组件注册表：espressif/brookesia_agent_openai
- 辅助头文件：`#include "brookesia/agent_helper/openai.hpp"`
- 辅助类：`esp_brookesia::agent::helper::Openai`

概述

brookesia_agent_openai 是基于 ESP-Brookesia Agent Manager 框架实现的 OpenAI 智能体。

功能特性

- **OpenAI Realtime API 集成**：通过 WebRTC 数据通道与 OpenAI 平台进行实时通信，支持语音对话和文本交互。
- **点对点通信**：基于 `esp_peer` 实现点对点（P2P）通信，支持信号通道和数据通道。
- **音频编解码**：内置音频编解码支持，默认使用 OPUS 格式，支持 16kHz 采样率和 24kbps 比特率。
- **数据通道消息处理**：支持多种数据通道消息类型，包括音频流、文本流、函数调用、会话管理等。
- **实时交互**：支持实时音频输入输出，提供低延迟的语音交互体验。
- **持久化存储**：可选搭配 *brookesia_service_nvs* 服务持久化保存配置信息。

服务接口

函数 无。

事件 无。

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

5.2.3 小智

- 组件注册表：espressif/brookesia_agent_xiaozhi
- 辅助头文件：`#include "brookesia/agent_helper/xiaozhi.hpp"`
- 辅助类：`esp_brookesia::agent::helper::XiaoZhi`

概述

brookesia_agent_xiaozhi 是为 ESP-Brookesia 生态系统提供的小智 AI 智能体实现。

功能特性

- **小智平台集成**：基于 `esp_xiaozhi` SDK 实现与小智 AI 平台的通信。
- **实时语音交互**：支持 OPUS 音频编解码，16kHz 采样率，24kbps 比特率。
- **丰富的事件支持**：支持说话/监听状态变化、智能体/用户文本、表情等事件。
- **手动监听控制**：支持手动开始/停止监听，适用于 Manual 对话模式。
- **中断说话**：支持中断智能体说话功能。
- **统一生命周期管理**：基于 `brookesia_agent_manager` 框架的统一智能体生命周期管理。

服务接口

函数

AddMCP_ToolsWithServiceFunction

描述 Add MCP tools from service functions. Return type: JSON array<string> of added tool names. Example: [“Service.Audio.SetVolume” ,” Service.Audio.GetVolume”]

执行要求

- 是否需要调度器: 不需要

参数

- `ServiceName`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Service name.
- `FunctionNames`
  - **类型**: `Array`
  - **是否必填**: 可选
  - **默认值**: `[]`
  - **描述**: Function names as JSON array<string>. Empty means all functions in the service. Example: [“SetVolume” ,” GetVolume”]

Schema JSON

CLI 命令

```
svc_call AgentXiaoZhi AddMCP_ToolsWithServiceFunction {"ServiceName":null,
↪"FunctionNames":null}
```

AddMCP_ToolsWithCustomFunction

描述 Add custom MCP tools. Return type: JSON array<string> of added tool names. Example: ["Custom.Display.GetBrightness" ," Custom.Display.SetBrightness"]

执行要求

- 是否需要调度器: 不需要

参数

- Tools
  - **类型**: Array
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call AgentXiaoZhi AddMCP_ToolsWithCustomFunction {"Tools":null}
```

RemoveMCP_Tools

描述 Remove MCP tools.

执行要求

- 是否需要调度器: 不需要

参数

- Tools
  - **类型**: Array
  - **是否必填**: 必填
  - **描述**: Tool names to remove. Example: ["Service.Audio.SetVolume","Custom.Display.GetBrightness"]

Schema JSON

CLI 命令

```
svc_call AgentXiaoZhi RemoveMCP_Tools {"Tools":null}
```

ExplainImage

描述 Explain an image. Return type: string. Example: "This image contains a cup on a desk."

执行要求

- 是否需要调度器: 需要

参数

- `Image`
  - **类型**: `RawBuffer`
  - **是否必填**: 必填
  - **描述**: Image data.
- `Question`
  - **类型**: `String`
  - **是否必填**: 可选
  - **默认值**: `"What is in the image?"`
  - **描述**: Question text.

Schema JSON

CLI 命令

```
svc_call AgentXiaoZhi ExplainImage {"Image":null,"Question":null}
```

事件

`ActivationCodeReceived`

描述 Emitted when an activation code is received.

执行要求

- 是否需要调度器: 需要

参数

- `Code`
  - **类型**: `String`
  - **描述**: Activation code.

Schema JSON

CLI 命令

```
svc_subscribe AgentXiaoZhi ActivationCodeReceived
```

相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

Chapter 6

AI 表达组件

本分类包含 ESP-Brookesia AI 表达相关组件的说明内容。

6.1 表情

- 组件注册表：espressif/brookesia_expression_emote
- 辅助头文件：`#include "brookesia/service_helper/expression/emote.hpp"`
- 辅助类：`esp_brookesia::service::helper::ExpressionEmote`

6.1.1 概述

brookesia_expression_emote 是 ESP-Brookesia 表情管理组件，基于 ESP-Brookesia 服务框架实现，提供：

- **资源管理**：支持加载和管理表情/动画资源，灵活配置资源。
- **表情控制**：支持设置和管理表情符号（emoji），用于表达不同的情感状态。
- **动画控制**：支持动画播放控制，包括等待动画帧完成、停止动画等操作。
- **二维码**：支持设置和隐藏二维码，实现二维码的显示和隐藏。
- **事件消息**：支持设置事件消息，实现事件消息的显示和隐藏。

6.1.2 服务接口

函数

`SetConfig`

描述 Set emote config.

执行要求

- 是否需要调度器: 不需要

参数

- `Config`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: 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 命令

```
svc_call Emote SetConfig {"Config":null}
```

LoadAssetsSource

描述 Load assets from the specified source.

执行要求

- 是否需要调度器: 不需要

参数

- `Source`
  - **类型**: `Object`
  - **是否必填**: 必填
  - **描述**: Asset source as a JSON object. Example: { “source” :” anim_icon” ,” type” :” PartitionLabel” ,” flag_enable_mmap” :false}

Schema JSON

CLI 命令

```
svc_call Emote LoadAssetsSource {"Source":null}
```

SetEmoji

描述 Set emoji and hide animation immediately.

执行要求

- 是否需要调度器: 不需要

参数

- `Emoji`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Emoji name.

Schema JSON

CLI 命令

```
svc_call Emote SetEmoji {"Emoji":null}
```

HideEmoji

描述 Hide current emoji.

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Emote HideEmoji
```

SetAnimation

描述 Set animation and hide emoji immediately.

执行要求

- 是否需要调度器: 不需要

参数

- `Animation`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Animation name.

Schema JSON

CLI 命令

```
svc_call Emote SetAnimation {"Animation":null}
```

InsertAnimation

描述 Insert animation; it hides immediately and shows after the duration.

执行要求

- 是否需要调度器: 不需要

参数

- `Animation`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Animation name.
- `DurationMs`
  - **类型**: `Number`
  - **是否必填**: 必填
  - **描述**: Animation duration in milliseconds. Stops automatically after this duration.

Schema JSON

CLI 命令

```
svc_call Emote InsertAnimation {"Animation":null,"DurationMs":null}
```

StopAnimation

描述 Stop current animation and hide it immediately.

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Emote StopAnimation
```

WaitAnimationFrameDone

描述 Wait for each animation frame to finish.

执行要求

- 是否需要调度器: 不需要

参数

- `TimeoutMs`
  - **类型**: `Number`
  - **是否必填**: 可选
  - **默认值**: `0E0`
  - **描述**: Timeout in milliseconds. *0* means wait forever.

Schema JSON

CLI 命令

```
svc_call Emote WaitAnimationFrameDone {"TimeoutMs":null}
```

`SetEventMessage`

描述 Set message for a specified emote event.

执行要求

- 是否需要调度器: 不需要

参数

- `Event`
  - **类型**: `String`
  - **是否必填**: 必填
  - **描述**: Event type. Allowed values: [Idle, Speak, Listen, System, User, Battery]
- `Message`
  - **类型**: `String`
  - **是否必填**: 可选
  - **默认值**: `""`
  - **描述**: Message text.

Schema JSON

CLI 命令

```
svc_call Emote SetEventMessage {"Event":null,"Message":null}
```

`HideEventMessage`

描述 Hide current event message.

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Emote HideEventMessage
```

SetQrcode

描述 Set QR code and hide emoji and animation immediately.

执行要求

- 是否需要调度器: 不需要

参数

- Qrcode
  - **类型**: String
  - **是否必填**: 必填
  - **描述**: QR code content.

Schema JSON

CLI 命令

```
svc_call Emote SetQrcode {"Qrcode":null}
```

HideQrcode

描述 Hide current QR code and show emoji immediately.

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Emote HideQrcode
```

NotifyFlushFinished

描述 Notify emote flush finished.

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Emote NotifyFlushFinished
```

`RefreshAll`

描述 Refresh the screen.

执行要求

- 是否需要调度器: 不需要

参数

- 无。

Schema JSON

CLI 命令

```
svc_call Emote RefreshAll
```

事件

`FlushReady`

描述 Emitted when emote flush is ready.

执行要求

- 是否需要调度器: 不需要

参数

- `Param`
  - **类型**: `Object`
  - **描述**: 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 命令

```
svc_subscribe Emote FlushReady
```

6.1.3 相关类型与配置

结构体、枚举、helper 方法以及完整 Doxygen API 请参考*helper 原始 API 参考*。

索引

符号

B

E

(*C++ enum*), 118

esp_brookesia::service::helper::ExpressionEmote::AssetServiceType::Max
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::AssetServiceType::PartitionLabel
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::AssetServiceType::Path
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::Config
(*C++ struct*), 118

esp_brookesia::service::helper::ExpressionEmote::Config::buf_pixels
(*C++ member*), 118

esp_brookesia::service::helper::ExpressionEmote::Config::flags.buff_dma
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::flags.buff_spiram
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::flags.double_buffer
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::flags.swap_color_bytes
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::fps
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::h_res
(*C++ member*), 118

esp_brookesia::service::helper::ExpressionEmote::Config::task_affinity
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::task_priority
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::task_stack
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::task_stack_in_ext
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::Config::v_res
(*C++ member*), 118

esp_brookesia::service::helper::ExpressionEmote::EventMessageType
(*C++ enum*), 117

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::Battery
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::Idle
(*C++ enumerator*), 117

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::Listen
(*C++ enumerator*), 117

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::Max
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::Speak
(*C++ enumerator*), 117

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::System
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::EventMessageType::User
(*C++ enumerator*), 118

esp_brookesia::service::helper::ExpressionEmote::FlashReadyEventParam
(*C++ struct*), 119

esp_brookesia::service::helper::ExpressionEmote::FlashReadyEventParam::data
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::FlashReadyEventParam::x_end
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::FlashReadyEventParam::x_start
(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::FlashReadyEventParam::y_end

(*C++ member*), 119

esp_brookesia::service::helper::ExpressionEmote::
(*C++ member*), 119

esp_brookesia::service::helper::NVS
(*C++ class*), 120

esp_brookesia::service::helper::NVS::DEFAULT_TIME
(*C++ member*), 123

esp_brookesia::service::helper::NVS::EntryInfo
(*C++ struct*), 123

esp_brookesia::service::helper::NVS::EntryInfo::k
(*C++ member*), 123

esp_brookesia::service::helper::NVS::EntryInfo::n
(*C++ member*), 123

esp_brookesia::service::helper::NVS::EntryInfo::t
(*C++ member*), 123

esp_brookesia::service::helper::NVS::erase_keys
(*C++ function*), 123

esp_brookesia::service::helper::NVS::EventId
(*C++ enum*), 120

esp_brookesia::service::helper::NVS::EventId::Max
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionEras
(*C++ enum*), 121

esp_brookesia::service::helper::NVS::FunctionEras
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionEras
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionGetP
(*C++ enum*), 121

esp_brookesia::service::helper::NVS::FunctionGetP
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionGetP
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionId
(*C++ enum*), 120

esp_brookesia::service::helper::NVS::FunctionId::
(*C++ enumerator*), 120

esp_brookesia::service::helper::NVS::FunctionId::
(*C++ enumerator*), 120

esp_brookesia::service::helper::NVS::FunctionId::
(*C++ enumerator*), 120

esp_brookesia::service::helper::NVS::FunctionId::
(*C++ enumerator*), 120

esp_brookesia::service::helper::NVS::FunctionId::
(*C++ enumerator*), 120

esp_brookesia::service::helper::NVS::FunctionList
(*C++ enum*), 121

esp_brookesia::service::helper::NVS::FunctionList
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionSetP
(*C++ enum*), 121

esp_brookesia::service::helper::NVS::FunctionSetP
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::FunctionSetP
(*C++ enumerator*), 121

esp_brookesia::service::helper::NVS::get_event_sc
(*C++ function*), 122

esp_brookesia::service::helper::NVS::get_function