鲁班 SDK

国产芯应用程序开发统一接口套件

User Manual

赋华慧创
iConnect Everything

目录

1 鲁班 SDK 介绍

1.1 鲁班 SDK 简介

鲁班 SDK 是一款面向国产化硬件平台的软硬件开发套件，由上海赋华慧创智能信息技术有限公司设计开发，专注于为国产操作系统（如统信 UOS、OpenEuler 等）和国产芯片（如海光、兆芯、飞腾、鲲鹏等）提供硬件驱动支持和标准化开发接口。它解决了国产化平台下外设驱动缺失、系统适配复杂等问题，帮助开发者快速构建本地化解决方案。

1.2 鲁班 SDK 特性

鲁班 SDK 主要特性包括：

1.开箱即用的硬件支持：预集成 Watchdog、GPIO、HWMON 等硬件驱动，提供标准操作接口，无需开发者深入硬件细节或从头开发驱动。

2.深厚的国产化适配：深度适配主流国产 OS（统信 UOS, OpenEuler, 麒麟等）和国产 CPU，解决了系统兼容性与稳定性问题。

3.极高的开发效率：提供简洁、清晰、稳定的 API 和大量即用型示例代码，显著降低开发难度和学习成本，加速项目上市时间。

4.功能全面，覆盖广泛：不仅提供硬件外设控制，还提供强大的系统级监控能力（LBTOP），满足工业应用中对系统健康状态的全面感知需求。

1.3 支持列表

鲁班 sdk 理论上支持各种 Linux X86 系统，目前我们测试验证过的平台如下：

操作系统	处理器架构
OpenEuler-22.03-LTS-SP4	IBZ70/IBZ50/IBH70/IBH71W/IBH71
Kylin-Desktop-V10-sp1-2403	IBZ70/IBZ50/IBH70/IBH71W/IBH71
UOS-desktop-20-professional-1070	IBZ70/IBZ50/IBH70/IBH71W/IBH71

2 鲁班 SDK 安装

2.1 鲁班 SDK 安装

➢ **安装前准备：**

系统要求：确保您的系统在支持列表之内。
权限要求：安装过程需要 root 权限。
依赖检查：通常 SDK 安装包会包含所有依赖，但建议系统已安装 gcc、make 等基础开发工具。

➢ **安装步骤：**

鲁班 SDK 提供了安装包，简单两步即可完成安装，安装按照如下步骤：
Step1:下载所需系统对应版本的鲁班 SDK 压缩包，解压压缩包。
tar -xzvf luban-sdk-kits-1.x.x-xxx.tar.gz
解压后目录下包括所有需要安装的 rpm/deb 包和 install.sh 脚本。
Step2：执行 install.sh 脚本一键安装所有安装包。
./install.sh
执行完成了鲁班 SDK 就已经成功安装了。在/usr/share/luban-sdk/example 目录下有我们提供的测试 Demo。

➢ **验证安装**

检查安装文件：
头文件：/usr/include/luban_sdk.h
库文件：/usr/lib/libluban_sdk.so (或类似路径)
示例程序：/usr/share/luban-sdk/example/
运行示例测试：
进入示例目录，运行测试程序，例如：
cd /usr/share/luban-sdk/example/
./hwm # 查看硬件监控信息

2.2 鲁班 SDK 更新

更新鲁班 SDK 之前，需要先卸载前期安装过的鲁班 SDK 的版本，然后再安装更新。以麒麟系统为例，luban-sdk-1.xxx.deb 包的卸载操作指令为：

dpkg -P luban-sdk

因鲁班 SDK 解压目录下除 luban-sdk-1.xxx.deb 外，可能还会有其他的 deb 包，需一起执行 dpkg -P xxx 进行卸载。如解压目录下存在的显卡驱动包 fantgpu-fh2m_3.xxx.deb，需执行 dpkg -P fantgpu-fh2m 进行卸载。

若鲁班 SDK 解压目录下，除 install.sh 外，其他包的后缀为.rpm，比如：适配 uos-server-20-1070e 或 OpenEuler-22.03-LTS-SP4 系统的鲁班 SDK，卸载指令需变

更为：

rpm -e luban-sdk 和 rpm -e fantgpu-fh2m。

卸载完成后可参考 2.1 章节中的安装步骤描述重新安装。

2.3 鲁班 SDK 信息查询

查询当前系统上安装的鲁班 SDK 的版本信息，以麒麟系统为例说明，可执行以下指令查询：

dpkg -l | grep luban-sdk

适配 uos-server-20-1070e 或 OpenEuler-22.03-LTS-SP4 系统的鲁班 SDK，查询指令为 rpm -qa | grep luban-sdk

相应的，查看芯动显卡驱动版本信息的指令为 dpkg -l | grep fantgpu 或 rpm -qa | grep fantgpu；查看网络驱动信息的指令为 modinfo ngbe；查看音频驱动信息的指令为 modinfo patch_senarytech

2.4 鲁班 SDK example 使用

进入/usr/share/luban-sdk/example 目录下，包括测试程序 gpio, watchdog, hwm, lbtop 等。

运行./watchdog –h 会看到 watchdog 测试程序具体使用以及参数介绍。

运行./gpio –h 会看到 gpio 测试程序具体使用以及参数介绍。

运行./lbtop –h 会看到 watchdog 测试程序具体使用以及参数介绍。

运行./hwm 会直接输出设备硬件电压电流风扇等硬件具体信息。

3 鲁班 SDK 功能介绍

鲁班 SDK 在 Driver 层提供硬件平台在指定 OS 中，缺少的各种外设驱动。这里主要支持的驱动列表及 API 主要功能如下：

模块	功能描述
GPIO	引脚初始化、输入/输出模式设置、电平读写
HWMON	硬件监控（电压、温度、风扇转速采集与配置）
Wactchdog	看门狗定时器初始化、超时设置、喂狗操作
MCU	读取硬件版本信息

鲁班 SDK 提供系统软硬件监控类 API 主要功能如下：

模块	功能描述
CPU	获取系统 CPU 详情、使用率、负载、时间统计
Disk	获取磁盘分区信息，读写速度统计
FS	获取挂载（指定）文件系统列表，文件系统总大小，已经空间，使用率
Memory	获取系统内存信息，内存使用百分比
SystemUptime	获取系统运行时间详情
Net	获取指定网络接口 I/O 统计，网络 I/O 速度
Proc	获取所有运行（指定）进程统计信息

鲁班 SDK 还提供 Demo 包括 watchdog ,gpio ,hwm ,lbtop 等具体用法参考 2.2 章节。

4 鲁班 SDK 开发

鲁班 SDK Library 层，针对二次开发需求，封装开放 API，访问外设，简化用户开发，降低开发难度。这里以 GPIO 开发为例简单说明如何开发。

第一步，鲁班 SDK 安装完成后，会在/usr/include 文件下生成 luban_sdk.h ，在/usr/lib 目录下生成 luban_sdk.so 文件。用户在开发自己程序时首先要在头文件中调用 luban_sdk.h

第二步，在调用 GPIO 相关 API 之前必须先初始化 SDK，具体参考如下：

```
#include <luban_sdk.h>

int main() {
    if (AdvSdkInit() != 0) {
        printf("SDK 初始化失败！ \n");
        return -1;
    }
    // 其他代码...
}
```

第三步，初始化 SDK 之后就可以使用 GPIO 相关 API，GPIO 部分也必须先初始化然后再执行其他设置或者获取 API，最后执行 UnInit 释放资源。如下是一个简单的开发示例：

```
#include <luban_sdk.h>
#include <stdio.h>

int main() {
    // 1. 初始化 SDK
    if (AdvSdkInit() != 0) {
        fprintf(stderr, "[ERROR] SDK init failed!\n");
        return -1;
    }

    // 2. 初始化 GPIO 引脚 12（输出模式）
    void* gpio = AdvGpioInit("/dev/gpiochip0", 12);
    if (!gpio) {
        fprintf(stderr, "[ERROR] GPIO init failed!\n");
        AdvSdkUnInit();  // 释放 SDK 资源
        return -1;
    }

    // 3. 设置高电平并读取状态
    AdvGpioDirectionOutput(gpio, 1);
    printf("GPIO12 value: %d\n", AdvGpioGetValue(gpio));

    // 4. 释放资源
```

```
    AdvGpioUnInit(gpio);
    AdvSdkUnInit();
    return 0;
}
```

Watchdog，hwm 等其他开发基本同上。用户可以参考我们提供的 example 进行自己代码开发。对于具体 API 介绍可以参考鲁班 SDK 中 DOC 目录下的关于 SDK 具体的说明。

5 鲁班 SDK 核心 API 详解

5.1 SDK 初始化与去初始化

int AdvSdkInit(void)
初始化 Luban SDK 运行环境，加载必要的驱动程序并完成相关设置工作。这是调用任何其他 SDK 功能的前提。
参数：无
返回值：
0 — 初始化成功。
非零值 — 初始化失败。
-5 — 不支持该 board。
void AdvSdkUnInit(void)
清理并释放由 AdvSdkInit 分配的所有资源。在程序退出前调用，以确保资源正确释放。
参数：无
返回值：无

5.2 GPIO 管理模块

用于控制通用输入输出引脚。
函数列表：

函数名	功能描述
void* AdvGpioInit(char *name, unsigned int gpio_line)	初始化 GPIO 引脚
int AdvGpioDirectionInput(void* handle)	设置 GPIO 为输入模式
int AdvGpioDirectionOutput(void* handle, unsigned int value)	设置 GPIO 为输出模式
int AdvGetGpioDirection(void* handle)	获取 GPIO 当前方向
int AdvGpioSetValue(void* handle, unsigned int value)	设置 GPIO 输出值
int AdvGpioGetValue(void* handle)	获取 GPIO 当前输入值
void AdvGpioUnInit(void* handle)	释放 GPIO 资源

函数详解：
void* AdvGpioInit(char *name, unsigned int gpio_line)
初始化指定的 GPIO 引脚，并返回一个操作句柄。
参数：

name — GPIO 设备名称，如 /dev/gpiochip0。
gpio_line — GPIO 引脚的索引号（Line Number）。
返回值：
成功 — 返回 GPIO 资源句柄 (void*)。
失败 — 返回 NULL。
备注：必须先调用此函数获得有效句柄，才能进行后续的 GPIO 操作。

int AdvGpioDirectionInput(void* handle)

将指定的 GPIO 引脚设置为输入模式。
参数：
handle — 由 AdvGpioInit 返回的 GPIO 资源句柄。
返回值：
0 — 成功。
-5 — 非法的 handle。
-1 — 失败，可通过 errno 查看具体错误原因。

int AdvGpioDirectionOutput(void* handle, unsigned int value)

将指定的 GPIO 引脚设置为输出模式，并设置其初始输出电平。
参数：
handle — 由 AdvGpioInit 返回的 GPIO 资源句柄。
value — 初始输出值：
0：低电平
1：高电平
返回值：
0 — 成功。
-5 — 非法的 handle。
-1 — 失败，可通过 errno 查看具体错误原因。

int AdvGetGpioDirection(void* handle)

获取指定 GPIO 引脚的当前方向配置。
参数：
handle — 由 AdvGpioInit 返回的 GPIO 资源句柄。
返回值：
0 — 输入模式。
1 — 输出模式。
-5 — 非法的 handle。
-1 — 失败，可通过 errno 查看具体错误原因。

int AdvGpioSetValue(void* handle, unsigned int value)

设置处于输出模式的 GPIO 引脚的输出电平。
参数：
handle — 由 AdvGpioInit 返回的 GPIO 资源句柄。
value — 要设置的输出值：
0：低电平
1：高电平
返回值：
0 — 成功。
-5 — 非法的 handle。

-1 — 失败，可通过 errno 查看具体错误原因。

int AdvGpioGetValue(void* handle)

获取 GPIO 引脚的当前电平值（无论方向是输入还是输出）。

参数：

handle — 由 AdvGpioInit 返回的 GPIO 资源句柄。

返回值：

0 — 低电平。

1 — 高电平。

-5 — 非法的 handle。

-1 — 失败，可通过 errno 查看具体错误原因。

void AdvGpioUnInit(void* handle)

释放由 AdvGpioInit 申请的资源。必须调用，否则会导致资源泄漏。

参数：

handle — 由 AdvGpioInit 返回的 GPIO 资源句柄。

返回值：无

GPIO 使用示例：

```
#include <luban_sdk.h>
#include <stdio.h>
#include <unistd.h> // for sleep

int main() {
    // 1. 初始化 SDK
    if (AdvSdkInit() != 0) {
        fprintf(stderr, "[ERROR] SDK init failed!\n");
        return -1;
    }

    // 2. 初始化 GPIO 引脚 12（假设连接 LED）
    void* gpio_led = AdvGpioInit("/dev/gpiochip0", 12);
    if (!gpio_led) {
        fprintf(stderr, "[ERROR] GPIO init failed!\n");
        AdvSdkUnInit();
        return -1;
    }

    // 3. 设置为输出模式，初始低电平
    if (AdvGpioDirectionOutput(gpio_led, 0) != 0) {
        fprintf(stderr, "[ERROR] Set GPIO output failed!\n");
        AdvGpioUnInit(gpio_led);
        AdvSdkUnInit();
        return -1;
    }
```

```
    // 4. 让 LED 闪烁 3 次
    for (int i = 0; i < 3; i++) {
        AdvGpioSetValue(gpio_led, 1); // LED 亮
        sleep(1);
        AdvGpioSetValue(gpio_led, 0); // LED 灭
        sleep(1);
    }

    // 5. 释放资源
    AdvGpioUnInit(gpio_led);
    AdvSdkUnInit();
    printf("GPIO example finished.\n");
    return 0;
}
```

5.3 HWMON 模块

用于监控主板上的硬件传感器信息，如电压、温度、风扇转速，并可配置风扇控制策略。

结构体详解：

struct Voltage

表示电压信息

```
struct Voltage {
    char Name[32];   // 电压名称 (e.g., "VCC", "3.3V")
    long Value;      // 当前电压值（单位：mV）
    long MinVal;     // 最小电压阈值（单位：mV）
    long MaxVal;     // 最大电压阈值（单位：mV）
};
```

struct Fan

表示风扇信息。

```
struct Fan {
    char Name[32];   // 风扇名称 (e.g., "FAN1", "CPU_FAN")
    long Value;      // 当前转速（单位：RPM）
    int index;       // 风扇索引号 (用于 AdvGet/SetHwmFanSettings)
};
```

struct Temperature

表示温度信息。

```
struct Temperature {
    char Name[32];   // 温度传感器名称 (e.g., "CPU", "SYS")
    long Value;      // 当前温度值（单位：毫摄氏度，即 1/1000 ° C）
};
```

enum pwm_enable

表示风扇控制模式的枚举。

```
enum pwm_enable {
```

```
    off = 0,                // 关闭模式（风扇全速运行）
    manual = 1,             // 手动模式（直接控制 PWM 占空比）
    thermal_cruise = 2, // 温度巡航模式（根据温度调整风扇）
    speed_cruise = 3,    // 转速巡航模式（维持目标转速）
    sf3 = 4,                // SMART FAN III 模式（已弃用或特定硬件支持）
    sf4 = 5                 // SMART FAN IV 模式（基于温度-PWM 映射的高级控制）
};
```

struct FanSettings

表示风扇的配置和监控设置。

```
struct FanSettings {
    unsigned char OutputType; // 风扇输出类型: 1=PWM, 0=DC
    unsigned char Mode;          // 控制模式: 使用 enum pwm_enable 的值
    int Target;                  // 目标值 (巡航模式下的目标温度或转速)
    int Tolerance;               // 目标值的容差
    int Temps[5];                // SMART FAN IV 模式的温度点 (毫摄氏度)
    int Pwms[5];                 // SMART FAN IV 模式的 PWM 占空比点 (0-255)
};
```

函数列表

函数名	功能描述
void* AdvHwmInit()	初始化硬件监控模块
int AdvGetHwmVoltages(void* h, struct Voltage **pv)	获取电压信息
int AdvGetHwmFans(void* h, struct Fan **pf)	获取风扇信息
int AdvGetHwmTemperatures(void* h, struct Temperature **pt)	获取温度信息
int AdvGetHwmFanSettings(void* h, int index, struct FanSettings *settings)	获取当前风扇设置
int AdvSetHwmFanSettings(void* h, int index, struct FanSettings settings)	配置风扇设置
void AdvHwmUninit(void* h)	释放硬件监控资源

函数详解

void* AdvHwmInit()

初始化硬件监控模块并返回资源句柄。

参数：无

返回值：

成功 — 返回硬件监控资源句柄 (void*)。

失败 — 返回 NULL。

int AdvGetHwmVoltages(void* h, struct Voltage **pv)

获取硬件监控器报告的所有电压信息。此函数会为 *pv 分配内存，调用者无需预先分配。
参数：
h — 由 AdvHwmInit 返回的硬件监控资源句柄。
pv — 指向 struct Voltage* 的指针。函数会使其指向新分配的内存块。
返回值：
成功 — 返回填充的 Voltage 结构体数量 (>0)。
失败 — 返回负的错误代码。
内存管理：此函数内部调用 malloc 分配内存。虽然后续调用 AdvHwmUninit 会释放这些资源，但最佳实践是在不再需要数据后立即手动释放：free(*pv);。

int AdvGetHwmFans(void* h, struct Fan **pf)

获取硬件监控器报告的所有风扇信息。内存管理同 AdvGetHwmVoltages。
参数：
h — 硬件监控资源句柄。
pf — 指向 struct Fan* 的指针。
返回值：
成功 — 返回填充的 Fan 结构体数量 (>0)。
失败 — 返回负的错误代码。

int AdvGetHwmTemperatures(void* h, struct Temperature **pt)

获取硬件监控器报告的所有温度信息。内存管理同 AdvGetHwmVoltages。
参数：
h — 硬件监控资源句柄。
pt — 指向 struct Temperature* 的指针。
返回值：
成功 — 返回填充的 Temperature 结构体数量 (>0)。
失败 — 返回负的错误代码。

int AdvGetHwmFanSettings(void* h, int index, struct FanSettings *settings)

从硬件监控设备中获取指定风扇的当前配置设置。
参数：
h — 硬件监控资源句柄。
index — 风扇的索引号（可从 struct Fan.index 获取）。
settings — 指向 FanSettings 结构体的指针，用于存储获取到的设置。
返回值：
0 — 成功。
负值 — 失败（错误代码）。
说明：如果 settings 指针为 NULL，函数不执行任何操作并返回 0。

int AdvSetHwmFanSettings(void* h, int index, struct FanSettings settings)

在硬件监控设备上配置指定风扇的设置。
参数：
h — 硬件监控资源句柄。
index — 风扇的索引号。
settings — 包含目标配置的 FanSettings 结构体。
返回值：
0 — 成功。

负值 — 失败（错误代码）。

说明：如果指定的 Mode 无效，函数可能会默认为手动模式 (1)。配置风扇需要相应的权限。

void AdvHwmUninit(void* h)

释放（反初始化）由 AdvHwmInit 返回的句柄及其相关的所有内部资源（包括由 AdvGetHwmVoltages/Fans/Temperatures 分配的内存）。

参数：

h — 硬件监控资源句柄。

返回值：无

HWMON 使用示例:

```
#include <luban_sdk.h>
#include <stdio.h>
#include <stdlib.h> // for free

void print_hwm_info() {
    void* hwm_handle = AdvHwmInit();
    if (!hwm_handle) {
        fprintf(stderr, "Failed to init HWM module.\n");
        return;
    }

    // 1. 获取并打印电压
    struct Voltage *volts = NULL;
    int volt_count = AdvGetHwmVoltages(hwm_handle, &volts);
    if (volt_count > 0) {
        printf("=== Voltages ===\n");
        for (int i = 0; i < volt_count; i++) {
            printf("%s: %.3fV (Min: %.3fV, Max: %.3fV)\n",
                   volts[i].Name,
                   volts[i].Value / 1000.0,
                   volts[i].MinVal / 1000.0,
                   volts[i].MaxVal / 1000.0);
        }
        free(volts); // 及时释放内存
    }

    // 2. 获取并打印温度
    struct Temperature *temps = NULL;
    int temp_count = AdvGetHwmTemperatures(hwm_handle, &temps);
    if (temp_count > 0) {
        printf("\n=== Temperatures ===\n");
        for (int i = 0; i < temp_count; i++) {
            printf("%s: %.1f°C\n", temps[i].Name, temps[i].Value / 1000.0);
```

```
        }
        free(temps);
    }

    // 3. 获取并打印风扇转速
    struct Fan *fans = NULL;
    int fan_count = AdvGetHwmFans(hwm_handle, &fans);
    if (fan_count > 0) {
        printf("\n=== Fans ===\n");
        for (int i = 0; i < fan_count; i++) {
            printf("%s: %ld RPM\n", fans[i].Name, fans[i].Value);
        }
        free(fans);
    }

    AdvHwmUninit(hwm_handle);
}

int main() {
    if (AdvSdkInit() != 0) {
        fprintf(stderr, "SDK init failed.\n");
        return -1;
    }

    print_hwm_info();

    AdvSdkUnInit();
    return 0;
}
```

5.4 WDT 模块

看门狗定时器用于在系统或程序异常挂起时自动复位设备，提高系统可靠性。
函数列表

函数名	功能描述
int AdvWDTInit()	初始化看门狗定时器
int AdvWDTSetConfig(int fd, int timeout)	设置看门狗超时时间
int AdvWDTGetConfig(int fd, int *timeout)	获取看门狗超时时间
int AdvWDTTrigger(int fd)	喂狗操作（重置定时器）

函数名	功能描述
int AdvWDTUninit(int fd)	关闭并释放看门狗资源

函数详解

int AdvWDTInit()

打开并初始化看门狗定时器设备。

参数：无

返回值：

成功 — 返回看门狗设备的文件描述符 (fd, >0)。

失败 — 返回 -1。

int AdvWDTSetConfig(int fd, int timeout)

设置看门狗定时器的超时时间。超时后如果不喂狗，系统将被复位。

参数：

fd — 由 AdvWDTInit 返回的文件描述符。

timeout — 超时时间，单位：秒。必须大于 0。

返回值：

0 — 成功。

-1 — 系统调用失败，检查 errno。

-2 — 参数无效 (fd <= 0 或 timeout <= 0)。

int AdvWDTGetConfig(int fd, int *timeout)

获取看门狗定时器当前的超时时间设置。

参数：

fd — 看门狗文件描述符。

timeout — 指向整数的指针，用于存储获取到的超时时间（秒）。

返回值：

0 — 成功。

负值 — 失败（错误代码）。

int AdvWDTTrigger(int fd)

向看门狗发送“喂狗”信号，重置其计时器，防止系统复位。需要在超时时间内定期调用。

参数：

fd — 看门狗文件描述符。

返回值：

0 — 成功。

负值 — 失败（错误代码）。

int AdvWDTUninit(int fd)

关闭看门狗定时器设备。调用此函数后，看门狗将停止工作，不再触发系统复位。

参数：

fd — 看门狗文件描述符。

返回值：

0 — 成功。

-1 — 无法禁用看门狗（例如写入 'V' 失败）。

-2 — 文件描述符无效 (fd <= 0)。

看门狗使用示例：

```
#include <luban_sdk.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    if (AdvSdkInit() != 0) {
        fprintf(stderr, "SDK init failed.\n");
        return -1;
    }

    // 1. 初始化看门狗
    int wdt_fd = AdvWDTInit();
    if (wdt_fd < 0) {
        fprintf(stderr, "WDT init failed.\n");
        AdvSdkUnInit();
        return -1;
    }

    // 2. 设置超时时间为 60 秒
    if (AdvWDTSetConfig(wdt_fd, 60) != 0) {
        fprintf(stderr, "Set WDT timeout failed.\n");
        close(wdt_fd); // 使用 close 而非 AdvWDTUninit
        AdvSdkUnInit();
        return -1;
    }

    printf("Watchdog started with 60s timeout. Press Ctrl+C to stop.\n");
    // 3. 主循环，定期喂狗
    while (1) {
        sleep(30); // 每 30 秒喂一次狗，远小于超时时间
        if (AdvWDTTrigger(wdt_fd) != 0) {
            fprintf(stderr, "Feed dog failed!\n");
            break;
        }
        printf("Dog fed.\n");
    }

    // 4. 程序正常结束，关闭看门狗
    AdvWDTUninit(wdt_fd);
    printf("Watchdog stopped.\n");

    AdvSdkUnInit();
```

```
    return 0;
}
```

5.4 MCU 模块

char* AdvReadMcuVersion()

读取并返回主板上 MCU（微控制单元）的版本信息字符串。

参数：无

返回值：

成功 — 返回指向版本信息字符串的指针（静态内存，无需释放）。

失败 — 返回 NULL。

示例：

```
#include <luban_sdk.h>
#include <stdio.h>
int main() {
    AdvSdkInit();
    char* ver = AdvReadMcuVersion();
    if (ver) {
        printf("MCU Version: %s\n", ver);
    } else {
        printf("Failed to read MCU version.\n");
    }
    AdvSdkUnInit();
    return 0;
}
```

5.5 LBTOP 系统监控模块

LBTOP 模块提供了一套全面的 API，用于监控系统的各项资源指标，包括 CPU、内存、磁盘、文件系统、网络、进程等。它是构建系统健康监控、性能分析工具的利器。

5.5.1 CPU 监控

结构体说明

CpuTimesStat

表示 CPU 时间统计信息。结构体中各个时间字段（如 User, Sys, Idle 等）单位为“jiffies”，而 CPUHz 表示的是每秒钟的“jiffies 数”（即 CPU 频率，单位 Hz）。所以在实际使用中，要将这些字段除以 CPUHz，才能换算成秒或其他标准时间单位。

```
struct CpuTimesStat {
    unsigned long long Flags;        // 位掩码标志，指示哪些字段有效
    unsigned long long Total;        // 所有 CPU 活动的总时间（jiffies）
```

```
    unsigned long long User;          // 用户模式时间（jiffies）
    unsigned long long Nice;          // 低优先级用户模式时间（jiffies）
    unsigned long long Sys;           // 系统（内核）模式时间（jiffies）
    unsigned long long Idle;          // 空闲时间（jiffies）
    unsigned long long Iowait;        // I/O 等待时间（jiffies）
    unsigned long long Irq;           // 硬件中断服务时间（jiffies）
    unsigned long long Softirq;       // 软件中断服务时间（jiffies）
    unsigned long CPUHz;                 // CPU 频率 (Hz)，用于 jiffies 转换
    int Ncores;                          // CPU 核心数
    unsigned long long XcpuTotal[MAX_CORES];    // 每个核心的总时间数组
    unsigned long long XcpuUser[MAX_CORES];     // 每个核心的用户模式时间数组
    // ... 其他核心-specific 的数组 (Nice, Sys, Idle, Iowait, Irq, Softirq)
};
```

CpuUsageRate

表示 CPU 使用率百分比。

```
struct CpuUsageRate {
    double User;      // 用户模式使用百分比(0.0 - 100.0)
    double Nice;      // 低优先级用户模式使用百分比
    double Sys;       // 系统模式使用百分比
    double Idle;      // 空闲百分比
    double Iowait;    // I/O 等待百分比
    double Irq;       // 硬件中断服务百分比
    double Softirq; // 软件中断服务百分比
};
```

CPUInfoStat

表示 CPU 核心的详细信息。

```
struct CPUInfoStat {
    int CPU;                 // 逻辑 CPU 索引
    char VendorID[16]; // 供应商 ID
    int Family;              // CPU 系列号
    int Model;               // 型号
    int Stepping;            // 步进号
    int PhysicalID;          // 物理封装 ID
    int CoreID;              // 物理核心 ID
    int Cores;               // 每个物理 CPU 的核心数
    char ModelName[64]; // 型号名称
    float MHz;               // 当前频率(MHz)
    long CacheSize;          // 缓存大小(KB)
    char Flags[256];         // 特性标志
    char Microcode[32]; // 微码版本
};
```

CPULoadavg

表示系统负载平均值。

```
struct CPULoadavg {
```

```
    double LoadAvg1Min;      // 1 分钟负载平均值
    double LoadAvg5Min;      // 5 分钟负载平均值
    double LoadAvg15Min;     // 15 分钟负载平均值
    int RunningTasks;        // 运行中的任务数
    int TotalTasks;          // 总任务数
    int LastPid;             // 最近创建的进程 PID
};
```

函数说明

int AdvGetCpuTimesStat(struct CpuTimesStat *stat)

收集高级 CPU 时间统计信息。

参数：

stat：指向要填充的 CpuTimesStat 结构体的指针。

返回值：

成功：解析的 CPU 核心数 (>0)。

失败：-1。

int AdvGetCpuUsage(struct CpuUsageRate* cpu, int milliseconds)

获取当前 CPU 使用百分比。此函数通过两次采样（间隔 milliseconds 毫秒）来计算使用率。

参数：

cpu：指向 CpuUsageRate 结构体的指针，用于接收结果。

milliseconds：采样间隔(毫秒)。必须大于 0。

返回值：

0：成功。

-1：获取 CPU 时间失败。

-2：时间间隔太短。

int AdvGetCPUInfoStat(struct CPUInfoStat* buf, int max)

获取 CPU 的详细信息。

参数：

buf：存储 CPU 信息的缓冲区。

max：缓冲区最大容量（能容纳的 CPUInfoStat 结构体数量）。

返回值：

成功：解析的逻辑 CPU 数 (>0)。

失败：-1。

int AdvGetCPULoadavg(struct CPULoadavg* loadavg)

获取 CPU 负载平均值。

参数：

loadavg：指向 CPULoadavg 结构体的指针。

返回值：

0：成功。

-1：失败。

double AdvGetCpuIdlePercent()

计算 CPU 总的空闲百分比。这是一个便捷函数。

返回值：

CPU 空闲百分比 (0.0 - 100.0)。

0.0：通常表示失败。

5.5.2 磁盘监控

结构体说明

PartStats

表示磁盘分区的 I/O 统计信息。

```
struct PartStats {
    char Name[32];              // 分区名称 (e.g., "sda1")
    unsigned long long XSectorsRead;   // 读取的扇区数
    unsigned long long XTimeReadMs;    // 读取时间(毫秒)
    unsigned long long XSectorsWrite; // 写入的扇区数
    unsigned long long XTimeWriteMs;   // 写入时间(毫秒)
};
```

DiskStats

表示物理磁盘及其分区的 I/O 统计信息。

```
struct DiskStats {
    char Name[32];                  // 磁盘名称 (e.g., "sda")
    char Type[16];                  // 磁盘类型 (e.g., "HDD", "SSD")
    unsigned long long SectorsRead; // 磁盘读取的总扇区数
    unsigned long long TimeReadMs;  // 磁盘读取的总时间(毫秒)
    unsigned long long SectorsWrite;// 磁盘写入的总扇区数
    unsigned long long TimeWriteMs; // 磁盘写入的总时间(毫秒)
    int PartCount;                  // 分区数
    struct PartStats Parts[16];     // 分区统计数组
};
```

DiskRWSpeed

表示磁盘读写速度统计。

```
struct DiskRWSpeed {
    char Name[32]; // 磁盘名称
    double RKbs;    // 读取速度(KB/s)
    double WKbs;    // 写入速度(KB/s)
};
```

函数说明

int AdvGetDiskStats(struct DiskStats* buf, int max, int all)

读取磁盘和分区 I/O 统计信息。

参数：

buf：存储磁盘统计信息的缓冲区。

max：缓冲区最大容量（能容纳的 DiskStats 结构体数量）。

all：是否包含所有设备（如 loopback 设备）。1=包含, 0=不包含。

返回值：

成功：填充的磁盘数量 (>0)。

失败：负值。

int AdvGetDiskRWSpeed(struct DiskRWSpeed* disk, int max, int milliseconds)

测量磁盘读写速度。通过两次采样（间隔 milliseconds 毫秒）来计算速度。

参数：

disk：存储速度结果的缓冲区。

max：缓冲区最大容量。

milliseconds：测量持续时间(毫秒)。最小为 50ms。

返回值：

成功：测量的磁盘数量 (>0)。

失败：负值。

5.5.3 文件系统监控

结构体说明

AdvFsUsage

表示文件系统使用统计。

```
struct AdvFsUsage {
    unsigned long long Blocks;      // 总块数
    unsigned long long Bfree;       // 超级用户可用块数
    unsigned long long Bavail;      // 普通用户可用块数
    unsigned long long Files;       // 总文件节点数
    unsigned long long Ffree;       // 空闲文件节点数
    unsigned long long BlockSize; // 块大小(字节)
};
```

MountEntry

表示挂载的文件系统信息。

```
struct MountEntry {
    dev_t Dev;                  // 设备 ID
    char DevName[64];           // 设备名称 (e.g., "/dev/sda1")
    char MountDir[256];         // 挂载目录 (e.g., "/")
    char Type[32];              // 文件系统类型 (e.g., "ext4")
};
```

函数说明

int AdvGetFsUsage(struct AdvFsUsage *buf, const char *path)

获取指定路径的文件系统使用统计。

参数：

buf：指向 AdvFsUsage 结构体的指针，用于存储结果。

path：文件系统挂载路径 (e.g., "/", "/home")。

返回值：

0：成功。

-1：失败。

uint64_t AdvFsTotalSize(const struct AdvFsUsage *fs)

计算文件系统总大小(字节)。便捷函数。

return fs->Blocks * fs->BlockSize;

uint64_t AdvFsFreeSize(const struct AdvFsUsage *fs)

计算非超级用户可用空间(字节)。便捷函数。

return fs->Bavail * fs->BlockSize;

uint64_t AdvFsUsedSize(const struct AdvFsUsage *fs)

计算已用空间(字节)。便捷函数。

return (fs->Blocks - fs->Bfree) * fs->BlockSize;

或 return AdvFsTotalSize(fs) - AdvFsFreeSize(fs);

double AdvFsUseRate(const struct AdvFsUsage *fs)

计算使用率(0.0 - 1.0)。便捷函数。

return (double)(fs->Blocks - fs->Bfree) / (double)fs->Blocks;

int AdvGetMountEntry(struct MountEntry* entries, int max, int all_fs)

获取挂载的文件系统列表。
参数：
entries：存储结果的缓冲区。
max：缓冲区最大容量。
all_fs：是否包含所有文件系统（如 proc, sysfs）。1=包含, 0=不包含。
返回值：
成功：挂载条目数 (>0)。
失败：-1。

int AdvGetAllMountedFsUsage(uint64_t* total, uint64_t* avail)

计算所有挂载文件系统的总空间和可用空间（仅统计实际存储设备，如 ext4, xfs 等）。
参数：
total：指向变量的指针，用于存储总空间(字节)。
avail：指向变量的指针，用于存储可用空间(字节)。
返回值：
>0：有效的文件系统数。
-1：获取挂载条目失败。
-2：无有效文件系统。

5.5.4 内存监控

结构体说明

MemoryUsage

表示内存使用摘要。

```
struct MemoryUsage {
    unsigned long long TotalKB;          // 总物理内存(KB)
    unsigned long long FreeKB;           // 完全空闲内存(KB)
    unsigned long long UsedKB;           // 已用内存(KB)
    unsigned long long BufferCacheKB; // 缓冲区/缓存内存(KB)
};
```

函数说明

int AdvGetMemoryUsage(struct MemoryUsage* mem)

获取当前系统内存使用摘要。

参数：

mem：指向 MemoryUsage 结构体的指针。

返回值：

0：成功。

double AdvMemUsedPercent(void)

计算内存使用百分比。便捷函数。

返回值：

内存使用百分比 (0.0 - 100.0)。

5.5.5 系统运行时间监控

结构体说明

SystemUptime

表示系统运行时间信息。

```
struct SystemUptime {
    double UpTime;      // 系统启动总时间(秒)
    double IdleTime;   // 所有 CPU 空闲总时间(秒)
    time_t BootTime;   // 启动时间(UNIX 时间戳)
};
```

函数说明

int AdvGetSystemUptime(struct SystemUptime* time)

获取系统运行时间详情。

参数：

time：指向 SystemUptime 结构体的指针。

返回值：

0：成功。

5.5.6 网络监控

结构体说明

NetIoStat

表示网络接口 I/O 统计和地址配置。

```
struct NetIoStat {
    unsigned int Flags;             // 接口标志
    unsigned int Mtu;               // MTU 大小
    struct in_addr Subnet;          // IPv4 子网掩码
    struct in_addr Address;         // IPv4 地址
    struct in6_addr Address6;       // IPv6 地址
    unsigned int Prefix6;           // IPv6 前缀长度
    unsigned int Scope6;            // IPv6 地址范围
    unsigned char HwAddress[6]; // MAC 地址
    // 各种数据包和字节计数器 (RxBytes, TxBytes, RxPackets, TxPackets, etc.)
    // 错误和丢弃数据包计数器 (RxErrors, TxErrors, RxDropped, TxDropped, etc.)
};
```

NetIoSpeed

表示网络 I/O 速度和错误统计。

```
struct NetIoSpeed {
    double RxSpeed;      // 接收速度(B/s)
    double TxSpeed;      // 发送速度(B/s)
    double RxDropRate; // 接收丢包率
    double TxDropRate; // 发送丢包率
    double RxErrRate;   // 接收错误率
    double TxErrRate;   // 发送错误率
};
```

函数说明

int AdvGetNetIoStat(const char *interface, struct NetIoStat* net)

获取指定网络接口的 I/O 统计和配置信息。

参数：

interface：网络接口名称 (e.g., "eth0", "wlan0")。

net：指向 NetIoStat 结构体的指针。

返回值：

0：成功。

<0：失败。

int AdvGetNetIoSpeed(const char *interface, int milliseconds, struct NetIoSpeed* speed)

计算网络 I/O 速度。通过两次采样（间隔 milliseconds 毫秒）来计算速度。

参数：

interface：网络接口名称。

milliseconds：测量间隔(毫秒)。最小为 50ms。

speed：指向 NetIoSpeed 结构体的指针。

返回值：

返回值：

0：成功。

负值：失败。

5.5.7 进程监控

结构体说明

ProcessStat

表示进程的详细统计信息。

```
struct ProcessStat {
    pid_t Pid;                  // 进程 ID
    uid_t Uid;                  // 用户 ID
    int Priority;               // 进程优先级
    int Nice;                   // nice 值
    char Cmdline[256];          // 命令行
    char State;                 // 进程状态 (R, S, D, Z, T, etc.)
    unsigned long long RChar;   // 读取的字符数 (来自存储)
    unsigned long long WChar;   // 写入的字符数 (到存储)
    unsigned long long Rss;     // 驻留集大小 (KB)
    unsigned long long Vsize;   // 虚拟内存大小 (KB)
    unsigned long long Utime;   // 用户模式 CPU 时间 (毫秒)
    unsigned long long Stime;   // 内核模式 CPU 时间 (毫秒)
    long long Starttime;        // 进程启动时间 (时钟滴答数 after boot)
};
```

函数说明

struct ProcessStat* AdvGetProcessStat(int* count)

获取所有运行进程的统计信息。此函数会分配内存，调用者负责释放。

参数：

count：指向整数的指针，用于存储获取到的进程数量。

返回值：

成功：指向 ProcessStat 结构体数组的指针。必须使用 free() 释放。

失败：NULL。

int AdvGetOneProcessStat(struct ProcessStat* ps, pid_t pid)

获取指定进程的统计信息。

参数：

ps：指向 ProcessStat 结构体的指针，用于存储结果。

pid：进程 ID。

返回值：

0：成功。

<0：失败。

5.5.8 LBTOP 系统监控模块示例

```
#include <luban_sdk.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
void print_system_status() {
    printf("===== System Status Overview =====\n");

    // 1. CPU 使用率
    struct CpuUsageRate cpu_usage;
    if (AdvGetCpuUsage(&cpu_usage, 500) == 0) { //  采样 500ms
        printf("CPU Usage: User=%.1f%%, Sys=%.1f%%, Idle=%.1f%%\n",
                cpu_usage.User, cpu_usage.Sys, cpu_usage.Idle);
    }

    // 2.  内存使用
    struct MemoryUsage mem;
    AdvGetMemoryUsage(&mem);
    printf("Memory: Total=%.2f MB, Used=%.2f MB (%.1f%%)\n",
            mem.TotalKB / 1024.0, mem.UsedKB / 1024.0, AdvMemUsedPercent());

    // 3.  磁盘使用  (根目录)
    struct AdvFsUsage fs;
    if (AdvGetFsUsage(&fs, "/") == 0) {
        printf("Root FS: Size=%.2f GB, Used=%.2f GB (%.1f%%)\n",
                AdvFsTotalSize(&fs) / (1024.0 * 1024.0 * 1024.0),
                AdvFsUsedSize(&fs) / (1024.0 * 1024.0 * 1024.0),
                AdvFsUseRate(&fs) * 100.0);
    }

    // 4.  负载平均值
    struct CPULoadavg load;
    if (AdvGetCPULoadavg(&load) == 0) {
        printf("Load Average: 1min=%.2f, 5min=%.2f, 15min=%.2f\n",
                load.LoadAvg1Min, load.LoadAvg5Min, load.LoadAvg15Min);
    }
    printf("==================================\n\n");
}

int main() {
    if (AdvSdkInit() != 0) {
        fprintf(stderr, "SDK init failed.\n");
        return -1;
    }

    //  每隔 2 秒打印一次系统状态
    for (int i = 0; i < 5; i++) {
        print_system_status();
```

```
            sleep(2);
        }

        AdvSdkUnInit();
        return 0;
}
```

上海赋华慧创智能信息技术科技有限公司

地址:上海市静安区江场三路 56、58 号 9 楼 902-1 室

网址: www.fuhuaxc.com

赋华慧创官方微信