LISTENAI

LISA_API 参考手册

文档密级：对外公开

Version 1.0 2021.1.04

声明

本手册由聆思科技版权所有，未经许可，任何单位和个人都不得以电子的、机械的、磁性的、光学的、化学的、手工的等形式复制、传播、转录和保存该出版物，或翻译成其它语言版本。一经发现，将追究其法律责任。

聆思科技保证本手册提供信息的准确性和可靠性。聆思科技保留更改本手册的权利，如有修改，恕不相告。请在订购时联系我们以获得产品最新信息。

对任何用户使用我们产品时侵犯第三方版权或其它权利的行为聆思科技概不负责。另外，在聆思科技未明确表示产品有该项用途时，对于产品使用在极端条件下导致一些失灵或损毁而造成的损失概不负责。

变更记录

版本	变更内容	变更人	审核人	日期
1.0	初稿	胡星晨	李逸卿	2020-1-04
1.1	优化文档描述，修改错别字	胡星晨	郝飞虎	2020-2-01

目录

概述

LISA_API 是聆思基于市场主流的 RTOS 和芯片平台，定义的一个介于底层驱动与上层应用的中间层架构。开发者需要基于 LISA_API ,对需要调用的硬件平台和实时操作系统接口抽象封装，这些抽象出的接口将为上层应用提供能力。

在接入 LISA_API 架构后，由于上层调用了 LISA_API 定义的标准接口，后续更换不同的芯片平台时，上层应用无需修改，只需做底层驱动的适配工作，这将减少平台迁移工作量，避免重复工作。

为了帮助开发者更好的理解 LISA_API，本文档将通过 Drivers（驱动）、OS（系统）、Modules（功能模块）三方面详细介绍 LISA_API。

注：以下接口使用用例基于 RTthread 平台，可移植于其他 RTOS 测试。

Drivers

封装 LISA 语音系统需要的基本驱动功能，输出统一 API

1. ADC 模块

1.1 简介

ADC 模块用于 LISA 系统的按键捕获功能中。故采用 LISA 语音系统的按键则需要实现该模块：

1.2 接口说明

```
lisa_err_t lisa_adc_get(uint8_t channel, uint32_t *data)
```

参数

channel　　adc 通道号(见 adc 简介)

*data　　获取 adc 值，单位 mv

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

获取对应通道号的 adc 采样值

1.3 使用示例

这是一个简单的 adc 程序使用示例，输入命令可以获取对应通道 adc 值

```
static int
lisa_adc(int argc, char *argv[])
{
uint32_t data = 0x00;
uint8_t ch;

if (argc < 2) {
LISA_LOGE(TAG, "usage: %s channel: ", argv[0]);
```

```
LISA_LOGE(TAG, "ADC_CHANNEL0_VBAT --- 0");
LISA_LOGE(TAG, "ADC_CHANNEL1_P4    --- 1");
LISA_LOGE(TAG, "ADC_CHANNEL2_P5    --- 2");
LISA_LOGE(TAG, "ADC_CHANNEL3_P23 --- 3");
LISA_LOGE(TAG, "ADC_CHANNEL4_P2    --- 4");
LISA_LOGE(TAG, "ADC_CHANNEL5_P3    --- 5");
LISA_LOGE(TAG, "ADC_CHANNEL6_P12 --- 6");
LISA_LOGE(TAG, "ADC_CHANNEL7_P13 --- 7");
return 0;
}

ch = atoi(argv[1]);
if (ch > 7) {
LISA_LOGE(TAG, "usage: %s channel: ", argv[0]);
LISA_LOGE(TAG, "ADC_CHANNEL0_VBAT --- 0");
LISA_LOGE(TAG, "ADC_CHANNEL1_P4    --- 1");
LISA_LOGE(TAG, "ADC_CHANNEL2_P5    --- 2");
LISA_LOGE(TAG, "ADC_CHANNEL3_P23 --- 3");
LISA_LOGE(TAG, "ADC_CHANNEL4_P2    --- 4");
LISA_LOGE(TAG, "ADC_CHANNEL5_P3    --- 5");
LISA_LOGE(TAG, "ADC_CHANNEL6_P12 --- 6");
LISA_LOGE(TAG, "ADC_CHANNEL7_P13 --- 7");

return 0;
}

lisa_adc_get(ch, &data);
LISA_LOGE(TAG, "ADC(%d) get %d", ch, data);
}

MSH_CMD_EXPORT(lisa_adc, flash testself);
```

2. I2S 模块

2.1 简介

I2S 模块用于 LISA 系统与 CSK 语音芯片的语音数据传输。注意该模块需要以阻塞方式实现。

2.2 接口说明

```
lisa_i2s_t *lisa_i2s_create(lisa_i2s_config_t *tconfig)
```

参数

tconfig i2s 配置参数

```
typedef struct {
    uint8_t index; //只有 1 个 i2s 模块，不用理会该序号
    uint8_t master; //主从模式选择
} lisa_i2s_params_t;

typedef struct {
    lisa_i2s_params_t paras; //i2s 配置参数:
    lisa_i2s_handler handler; //暂未支持
} lisa_i2s_config_t;
```

返回值

成功返回 i2s 设备句柄；失败返回 NULL

说明

初始化 i2s 配置，目前只支持主从配置，其他参数配置固定写在驱动中。

采样率：16000

位宽：32

模式：I2S_LEFT_JUSTIFIED，即标准 i2s 不偏移 1bclk

大小端：MSB

```
lisa_err_t lisa_i2s_enable(lisa_i2s_t *dev)
```

参数

dev i2s 设备句柄

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

使能 i2s 接收 DMA，调用该函数后，底层驱动会开始接收数据并存入 ringbuf 中，待用户调用 lisa_i2s_read 获取数据

```
lisa_err_t lisa_i2s_disable(lisa_i2s_t *dev)
```

参数

dev　　i2s 设备句柄

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

关闭 i2s 接收 DMA，使用之前需 lisa_i2s_create 获取设备句柄

```
int32_t lisa_i2s_read(lisa_i2s_t *dev, void *data, uint16_t len)
```

参数

dev　　i2s 设备句柄

data　　接收数据的 buf

len　　需要接收的数据长度

返回值

实际接收到的数据长度

说明

该函数是使用阻塞的方式，调用时如果底层驱动未接收到数据会阻塞当前线程

```
int32_t lisa_i2s_write(lisa_i2s_t *dev, void *data, uint16_t len)
```

功能未开放

2.3 使用示例

```
static void i2s_slave_thread(void *parameter){
    lisa_i2s_config_t i2s_config;
    i2s_config.handler = i2s_handler_test;
    i2s_config.paras.master = 0;
    g_i2s_dev = lisa_i2s_create(&i2s_config); //初始化 i2s 为从模式
```

```
    lisa_i2s_enable(g_i2s_dev); //使能 DMA

    uint32_t i;
    while (1) {
        size = lisa_i2s_read(g_i2s_dev, rxbuf, 512); //读取数据
... ...
    }
}
```

3. GPIO 模块

3.1 简介

通用输入/输出模块，可以通过相关接口完成 IO 配置和控制。

3.2 接口说明

```
lisa_err_t lisa_gpio_init(lisa_gpioconfig_t *tconfig)
```

参数

tconfig　　gpio 配置

```
typedef enum {
    GPIO_MODE_OUTPUT = 0,                   //输出模式
    GPIO_MODE_INPUT_PULLUP,                 //输入下拉
    GPIO_MODE_INPUT_PULLDOWN,  //输入上拉

    PIN_MODE_INVALID,
} lisa_gpio_mode_e;

typedef enum {
    GPIO_TRIG_RISING = 0,           //上升沿触发
    GPIO_TRIG_FALLING,                      //下降沿触发
```

```
    // GPIO_TRIG_BOTH,          //bk7251 不支持

    GPIO_TRIG_INVALID,
} lisa_gpio_trig_e;

typedef struct {
uint8_t num;                                    //引脚号
lisa_gpio_mode_e mode;                          //引脚模式
lisa_gpio_trig_e trig;                          //触发方式
lisa_gpio_handler handler;                      //注册回调
} lisa_gpioconfig_t;
```

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

引脚使用前需要设置好模式，触发方式以及中断回调。如果无需中断模式，trig 设置为 GPIO_TRIG_INVALID，handler 设置为 NULL 即可。

```
lisa_err_t lisa_gpio_set(uint8_t num, uint8_t value)
```

参数

num 引脚号

value 输出电平

0 低电平

1 高电平

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

设置 GPIO 输出对应电平，使用前需设置为 GPIO_MODE_OUTPUT

```
uint8_t lisa_gpio_get(uint8_t num)
```

参数

num 引脚号

返回值

引脚电平值 0 低电平

1 高电平

说明

获取 GPIO 当前电平。

```
lisa_err_t lisa_gpio_enable(uint8_t num)
```

参数

num 引脚号

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

使能 GPIO 中断，使用前需设置好触发方式 trig 和回调函数 handler。

```
lisa_err_t lisa_gpio_disable(uint8_t num)
```

参数

num 引脚号

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

关闭 GPIO 中断，使用前需设置好触发方式 trig 和回调函数 handler。

3.3 使用示例

```
uint8_t opt, num, value;
```

```
void
gpio_handler(void *para)
{
        LISA_LOGE(TAG, "detect button %d !", lisa_gpio_get(num));
}
static int
GPIO_Test(int argc, char *argv[])
{
        LISA_LOGE(TAG, "GPIO_Test !");

        if (argc != 4) {
                LISA_LOGE(TAG, "cmd err !");
                return 0;
        }

        opt = atoi(argv[1]);
        num = atoi(argv[2]);
        value = atoi(argv[3]);

        LISA_LOGE(TAG, "opt:%d    num:%d value:%d !", opt, num, value);

        if (opt) {
                lisa_gpioconfig_t tConfig;

                tConfig.num = num;
                tConfig.mode = GPIO_MODE_OUTPUT;
```

```
                tConfig.trig = GPIO_TRIG_INVALID;
                tConfig.handler = NULL;
                lisa_gpio_init(&tConfig);

                lisa_gpio_set(num, value);
        } else {
                lisa_gpioconfig_t tConfig;

                tConfig.num = num;
                tConfig.mode = GPIO_MODE_INPUT_PULLDOWN;
                tConfig.trig = GPIO_TRIG_RISING;
                tConfig.handler = gpio_handler;
                lisa_gpio_init(&tConfig);

#if 1
                lisa_err_t ret;
                ret = lisa_gpio_enable(num);
                if (ret != LISA_OK) {
                        LISA_LOGE(TAG, "lisa_gpio_enable err !");
                }
#endif

                LISA_LOGE(TAG, "get value %d !", lisa_gpio_get(num));
        }

        return 0;
```

```
}


MSH_CMD_EXPORT(GPIO_Test, gpio testself);
```

4 . PWM 模块

4.1 简介

PWM 模块用于 LISA 系统的灯光实现中，一种灯光颜色需要实现一组 PWM。

4.2 接口说明

```
lisa_pwm_t *lisa_pwm_create(lisa_pwm_config_t *pconfig)
```

参数

pconfig PWM 配置

```
typedef struct {
    uint8_t pwm_index; //通道号
} lisa_pwm_config_t;

typedef struct {
    lisa_pwm_config_t config;
} lisa_pwm_t;
```

返回值

PWM 设备句柄

说明

使用之前完成通道号对应 PWM 初始化

```
lisa_err_t lisa_pwm_set(lisa_pwm_t *dev, uint32_t frequency, uint16_t duty)
```

参数

dev 设备句柄

frequency 频率

duty 占空比

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

完成 PWM 的配置

```
lisa_err_t lisa_pwm_start(lisa_pwm_t *dev)
```

参数

dev 设备句柄

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

启动 PWM

```
lisa_err_t lisa_pwm_stop(lisa_pwm_t *dev)
```

参数

dev 设备句柄

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

停止 PWM

4.3 使用示例

```
lisa_pwm_t *g_pwm_test;


static int
lisa_pwm(int argc, char *argv[])
{
```

```
        if (argc < 2) {
                LISA_LOGE(TAG, "pwm: %s ch duty cycle ", argv[0]);
                return 0;
        }

        uint8_t ch = atoi(argv[1]);
        uint32_t duty = atoi(argv[2]);
        uint32_t cycle = atoi(argv[3]);
        lisa_pwm_config_t tconfig;

        LISA_LOGE(TAG, "pwm %d %d %d", ch, duty, cycle);

        tconfig.pwm_index = ch;
        g_pwm_test = lisa_pwm_create(&tconfig);

        lisa_pwm_set(g_pwm_test, cycle, duty);

        lisa_pwm_start(g_pwm_test);
}

MSH_CMD_EXPORT(lisa_pwm, pwm testself);
```

5. RTC 模块

5.1 简介

用于和云端同步时间

5.2 接口说明

```
lisa_err_t lisa_rtc_set(lisa_time_t *time)
```

参数

time 设置时间的数据指针

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

设置时间

```
lisa_err_t lisa_rtc_get(lisa_time_t *time)
```

参数

time 获取时间的数据指针

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

获取时间

5.3 使用示例

```
static int
rtc_test(int argc, char *argv[])
{
        lisa_time_t time;
        uint8_t set = 0;

        if (argc < 2) {
                LISA_LOGE(TAG, "usage: %s set|get\r\n", argv[0]);
```

```
                return 0;
        }

        lisa_rtc_get(&time);
        if ('s' == argv[1][0]) {
                set = 1;
        } else {
                set = 0;
        }

        if (set) {
                time.year = 2013;
                time.month = 11;
                time.day = 12;
                time.hour = 13;
                time.minute = 14;
                time.second = 15;
                lisa_rtc_set(&time);
        } else {
                lisa_rtc_get(&time);
        }
}

MSH_CMD_EXPORT(rtc_test, rtc testself);
```

6. UART 模块

6.1 简介

UART 模块用于 LISA 系统和 CSK 语音芯片通讯控制。该模块需以非阻塞方式实现 UART 的接受和发送，且支持 中断回调。

6.2 接口说明

```
lisa_uart_t *lisa_uart_create(lisa_uart_config_t *tconfig)
```

参数

tconfig　　uart 配置参数

```
//中断回调事件
#define LISA_UART_RECEIVEDONE (1UL << 0) //接收到数据或者空闲超时
#define LISA_UART_SENDDONE (1UL << 1) //数据发送完毕


typedef struct {
    uint8_t index; // 1-uart1 2-uart2
    uint8_t rate; //暂时不可配置,默认使用 115200
} lisa_uart_params_t;

typedef struct {
    lisa_uart_params_t paras; //配置参数
    lisa_uart_handler handler; //中断回调，目前支持
LISA_UART_RECEIVEDONE/LISA_UART_SENDDONE
} lisa_uart_config_t;
```

返回值

成功返回 I2S 设备句柄；　失败返回 NULL

说明

完成 UART 配置获取设备句柄

```
lisa_err_t lisa_uart_enable(lisa_uart_t *dev)
```

参数

dev 设备句柄

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

使能 uart 中断

```
lisa_err_t lisa_uart_disable(lisa_uart_t *dev)
```

参数

dev 设备句柄

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

关闭 uart 中断

```
int32_t lisa_uart_send(lisa_uart_t *dev, const uint8_t *data, uint16_t len)
```

参数

dev 设备句柄

data 待发送的数据 buf

len 待发送的数据长度

返回值

实际发送的数据长度

说明

该函数采用非阻塞方式，可使用中断回调异步通知是否发送完毕

```
int32_t lisa_uart_receive(lisa_uart_t *dev, uint8_t *data, uint16_t len)
```

参数

dev 设备句柄

data 待接收的数据 buf

len 待接收的数据长度

返回值

实际接收的数据长度

说明

该函数采用非阻塞方式，可使用中断回调异步通知是否接收到数据或者空闲超时。

6.3 使用示例

```
lisa_uart_t *uart_dev;

static rt_mq_t rx_msg;
static rt_mq_t tx_msg;

static uint8_t rec_fifo[128];
static uint8_t tx_fifo[128];
static uint8_t ringbuf[512];
static struct rt_ringbuffer rb;

static void
serial_rx_entry(void *parameter)
{
uint32_t reclen;
uint32_t ret;
uint32_t recvcnt;
uint32_t cnt = 0;
LISA_LOGE(TAG, "serial_rx_entry");

while (1) {
rt_mq_recv(rx_msg, &recvcnt, sizeof(uint32_t), RT_WAITING_FOREVER);

reclen = lisa_uart_receive(uart_dev, rec_fifo, recvcnt);
while (reclen != 0) {
ret = rt_ringbuffer_put(&rb, rec_fifo, reclen);
if (ret != reclen) {
LISA_LOGE(TAG, "rec ringbuf over!");
} else {
#if 0
LISA_LOGE(TAG, "afterringbuf rec%d", ret);
for (uint8_t i = 0; i < 10; i++) {
LISA_LOGE(TAG, "%d", *(rec_fifo + i));
}
#endif
```

```
}
cnt += ret;
rt_mq_send(tx_msg, &ret, sizeof(uint32_t));
reclen = lisa_uart_receive(uart_dev, rec_fifo, recvcnt);
}
cnt = 0;
}
}

static void
serial_tx_entry(void *parameter)
{
uint32_t ret;
uint32_t len;

LISA_LOGE(TAG, "serial_tx_entry");

while (1) {
rt_mq_recv(tx_msg, &len, sizeof(uint32_t), RT_WAITING_FOREVER);

ret = rt_ringbuffer_get(&rb, tx_fifo, len);
if (ret != len) {
LISA_LOGE(TAG, "get ringbuf lose %d %d!", ret, len);
}

lisa_uart_send(uart_dev, tx_fifo, len);

LISA_LOGE(TAG, "send%d", len);
}
}

void
uart_handler(uint32_t event, void *para)
{
uint32_t size = *(uint32_t *)para;

switch (event) {
case LISA_UART_RECEIVEDONE:
rt_mq_send(rx_msg, &size, sizeof(uint32_t));
LISA_LOGE(TAG, "LISA_UART_RECEIVEDONE %d!", size);
break;

case LISA_UART_SENDDONE:
LISA_LOGE(TAG, "LISA_UART_SENDDONE");
```

```
break;

default:
break;
}
// rt_sem_release(&rx_sem);
}

static int
uartfifo_test(int argc, char *argv[])
{
if (argc != 2) {
LISA_LOGE(TAG, "cmd err !");
return 0;
}

uint8_t uartindex;
uartindex = atoi(argv[1]);
LISA_LOGE(TAG, "uartfifo_test %d!", uartindex);

rt_ringbuffer_init(&rb, ringbuf, sizeof(ringbuf));
rx_msg = rt_mq_create("rx_msg", sizeof(uint32_t), 10, RT_IPC_FLAG_FIFO);
tx_msg = rt_mq_create("tx_msg", sizeof(uint32_t), 10, RT_IPC_FLAG_FIFO);

lisa_uart_config_t tconfig;
tconfig.paras.index = uartindex;
tconfig.handler = uart_handler;

uart_dev = lisa_uart_create(&tconfig);
if (!uart_dev) {
LISA_LOGE(TAG, "lisa_uart_init fail!");
}

rt_thread_t thread = rt_thread_create("serial", serial_rx_entry, RT_NULL, 1024, 25, 10);
if (thread != RT_NULL) {
rt_thread_startup(thread);
} else {
LISA_LOGE(TAG, "rt_thread_create err!");
}

thread = rt_thread_create("serial", serial_tx_entry, RT_NULL, 1024, 25, 10);
if (thread != RT_NULL) {
rt_thread_startup(thread);
} else {
```

```
LISA_LOGE(TAG, "rt_thread_create err!");
}

char str[] = "uartfifo_test send!\r\n";
lisa_uart_send(uart_dev, str, (sizeof(str) - 1));
return 0;
}
MSH_CMD_EXPORT(uartfifo_test, uart testself);
```

7 .FLASH 模块

7.1 简介

FLASH 模块用于支持 LISA 的持久化功能模块，以及对固化在 FLASH 中的音频文件的读取操作。

7.2 接口说明

```
lisa_err_t lisa_flash_write(uint32_t addr, uint8_t *buf, uint32_t size)
```

参数

addr 待写入的起始地址

buf 待写入的数据 buf 指针

size 待写入的数据长度

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

FLASH 写入操作，写之前需要先擦除。

```
lisa_err_t lisa_flash_read(uint32_t addr, uint8_t *buf, uint32_t size)
```

参数

addr 待读取的起始地址

buf 待读取的数据 buf 指针

size 待读取的数据长度

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

FLASH 读取操作

```
lisa_err_t lisa_flash_erase(uint32_t addr, uint32_t size)
```

参数

addr 待擦除的起始地址

size 待擦除的数据长度

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

FLASH 擦除，最小单位是 4K

7.3 使用示例

```
uint8_t buf[4096];

static int

lisa_flash(int argc, char *argv[])

{

        uint32_t address, len;

        void *data;


        if (argc < 4) {

                LISA_LOGE(TAG, "usage: %s write/read/erase addr len\r\n", argv[0]);

                return 0;

        }
```

```
address = atoi(argv[2]);
len = atoi(argv[3]);
if (len > sizeof(buf)) {
        len = sizeof(buf);
}

LISA_LOGE(TAG, "address 0x%x    len:%d\r\n", address, len);

// bk_flash_enable_security(FLASH_UNPROTECT_LAST_BLOCK);

LISA_LOGE(TAG, "protect:%d\r\n", get_flash_protect());

if (strcmp(argv[1], "write") == 0) {
        for (uint8_t i = 0; i < len; i++) {
                buf[i] = i;
        }

        lisa_flash_write(address, buf, len);
} else if (strcmp(argv[1], "read") == 0) {
        for (uint8_t i = 0; i < len; i++) {
                buf[i] = 0;
        }

        lisa_flash_read(address, buf, len);

        for (uint8_t i = 0; i < len; i++) {
```

```
                LISA_LOGE(TAG, "rec:0x%x\r\n", buf[i]);
            }
        } else if (strcmp(argv[1], "erase") == 0) {
            lisa_flash_erase(address, 4096);
        }
}

MSH_CMD_EXPORT(lisa_flash, flash testself);
```

8. WiFi 模块

8.1 简介

提供 API 实现 WiFi 的 AP 模式和 STA 模式的使能和除能，实现 WiFi 参数的获取，并可根据回调函数返回 WiFi 的事件中断。

8.2 接口说明

接口

```
lisa_err_t lisa_wifi_init(void);
```

参数

none

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

通过该函数上电 WiFi 的硬件。

接口

```
lisa_err_t lisa_wifi_connect(lisa_wifi_connect_t *connect);
```

参数

```
typedef struct {

char *ssid; // 连入的 ssid

char *password; // 连入的密码，无密码置 NULL

uint8_t *bssid;                    // 不限定则传 NULL

void (*on_event)(lisa_wifi_connect_event_t *event); // 注册 wifi 事件回调函数

} lisa_wifi_connect_t;
```

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

该函数启动 WiFi 的 STA 模式，并连接到指定的 WiFi 热点。（仅在当前 WiFi 模式为空时生效）

接口

```
lisa_err_t lisa_wifi_disconnect(void);
```

参数

none

*返回值**

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

该函数断开 WiFi 连接，AP 模式和 STA 模式均生效。

接口

```
lisa_err_t lisa_wifi_start_ap(lisa_wifi_ap_t *ap);
```

参数

```
typedef struct {
```

```
    char ssid[33];                  // 创建的热点名称

    char password[64];              // 创建的热点密码，为空则置 NULL

    uint8_t ch;                     // 创建的热点信道号，默认为 11 信道，非法值暂不提醒

    uint8_t ip_addr[4];     //  创建的热点 IP 地址（暂未实现）

    uint8_t subnet_mask[4];     //  创建的热点 IP 地址掩码（暂未实现）

    uint8_t client_ip_start[4];     //  创建的热点可分配的 IP 起始地址（暂未实现）

    uint8_t client_ip_end[4];       //  创建的热点 IP 地址（暂未实现）

} lisa_wifi_ap_t;
```

*返回值**

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

该函数启动 WiFi 的 AP 模式，并创建指定的热点。

接口

```
lisa_err_t lisa_wifi_stop_ap(void);
```

参数

none

*返回值**

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

该函数关闭 wifi 的 ap 模式。

暂未实现的预留接口

开始搜索周围环境的 WiFi 信号

```
lisa_err_t lisa_wifi_start_scan(lisa_wifi_scan_t *scan);
```

停止搜索周围环境的 WiFi 信号

```
lisa_err_t lisa_wifi_stop_scan(void);
```

8.3 使用示例

启动 WiFi 功能：>> hal_wifi_demo init

启动 STA 模式并连入 WiFi：>> hal_wifi_demo join "wifi_ssid " "wifi_password"

启动 AP 模式并创建 WiFi 热点：>> hal_wifi_demo ap "wifi_ssid " "wifi_password"

关闭当前 WiFi 模式 hal_wifi_demo down

```
static void
wifi_event_callback(lisa_wifi_connect_event_t *event)
{
WLAN_DBG("------------event type: [%d]---------------",event->what);
}

int
hal_wifi_demo(int argc, char **argv)
{
  if (strcmp(argv[1], "init") == 0) {
  lisa_wifi_init();
  } else if (strcmp(argv[1], "join") == 0) {
        lisa_wifi_connect_t test_connect;
        test_connect.ssid = argv[2];

        test_connect.password = argv[3];
        test_connect.on_event = wifi_event_callback;
        lisa_wifi_connect(&test_connect);
} else if (strcmp(argv[1], "up") == 0) {
```

```
/* the key was saved in g_lisa_wlan device */
rt_wlan_connect(lisa_wifi_info->g_lisa_wlan, RT_NULL, lisa_wifi_info->g_lisa_wlan->key);
} else if (strcmp(argv[1], "down") == 0) {
lisa_wifi_disconnect();
} else if (strcmp(argv[1], "ap") == 0) {
rt_err_t result = RT_EOK;
lisa_wifi_ap_t temp_ap;
        os_memset(temp_ap.ssid, 0, sizeof(temp_ap.ssid));
        os_memset(temp_ap.password, 0, sizeof(temp_ap.password));
        lisa_wifi_info->connect_event.on_event = NULL;
        os_memcpy(temp_ap.ssid, argv[2], strlen((char *)argv[2]) + 1);

        if (argc == 4) {
        } else if (argc == 5) {
        os_memcpy(temp_ap.password, argv[3], strlen((char *)argv[3]) + 1);
        }
        temp_ap.ch = 11;
        result = lisa_wifi_start_ap(&temp_ap);
        if (result != RT_EOK) {
        WLAN_DBG("wifi start failed! result=%d\n", result);
    }
}
return 0;
}

MSH_CMD_EXPORT(hal_wifi_demo, hal_wifi_demo command);
```

OS

OS 的 API 函数为应用层提供了统一的函数接口；目的是屏蔽不同 OS 的 API 的差别，方便应用层代码开发和移植

1. Thread

lisa_thread_create

```
lisa_thread_t *lisa_thread_create(const lisa_thread_attr_t *attr,
void (*entry)(void *),
void *arg)
```

参数：

attr：　　线程属性

```
typedef struct {
    char *name;
    uint32_t stack_size; // 栈大小
    uint32_t priority; // 优先级
} lisa_thread_attr_t;
```

entry：　　线程函数

```
void (*entry)(void *)
```

arg：　　线程函数的参数

返回值：

lisa_thread_t *　线程指针

失败　返回 NULL

成功　返回非 NULL

说明：

创建线程

lisa_thread_delete

```
lisa_err_t lisa_thread_delete(lisa_thread_t *thread)
```

参数：

thread： 线程指针

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

销毁线程

lisa_thread_yield

```
lisa_err_t lisa_thread_yield(void)
```

参数：

void

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

启动线程调度

lisa_thread_suspend

```
lisa_err_t lisa_thread_suspend(lisa_thread_t *thread)
```

参数：

thread： 线程指针

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

挂起线程

lisa_thread_resume

```
lisa_err_t lisa_thread_resume(lisa_thread_t *thread)
```

参数：

thread： 线程指针

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

恢复线程

lisa_thread_delay

```
lisa_err_t lisa_thread_delay(uint32_t ticks)
```

参数：

tick： 延时时间，单位为秒

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

线程延时，单位为秒

lisa_thread_mdelay

```
lisa_err_t lisa_thread_mdelay(uint32_t ms)
```

参数：

ms： 单位为 ms

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

线程延时，单位为 ms

lisa_thread_getcurrent

```
lisa_thread_t *lisa_thread_getcurrent(void)
```

参数：

void

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

获取当前线程指针

2. Queue

lisa_queue_create

```
lisa_queue_t *lisa_queue_create(uint32_t count, uint32_t item_size)
```

参数：

count： 消息队列中消息的最大个数

item_size： 单个消息的大小

返回值：

lisa_queue_t *：消息队列指针

成功： 返回消息队列指针

失败： 返回 NULL

说明：

创建消息队列

lisa_queue_push

```
lisa_err_t lisa_queue_push(lisa_queue_t *queue,
                           void *item,
                           uint32_t item_size,
                           int32_t wait)
```

参数：

queue： 消息队列指针

item： 发送数据的地址

item_size： 单个消息的大小

wait： 超时时间，单位为 ms

返回值：

LISA_OK： 发送成功

LISA_FAIL： 出现错误

说明：

向消息队列里面发送消息

lisa_queue_receive

```
lisa_err_t lisa_queue_receive(lisa_queue_t *queue,
```

```
                                                    void *item,
                                                    uint32_t
item_size,
                                                    int32_t wait)
```

参数：

queue：　　消息队列指针

item：　　接收数据的指针

item_size：　单个消息的大小

wait：　　超时时间，单位为 ms

返回值：

LISA_OK：　接收成功

LISA_FAIL：　出现错误

说明：

从消息队列里面接收消息；如果超时时间到了，还没有收到消息，那么返回 LISA_FAIL；在超时时间到达前收到消息，那么返回 LISA_OK；

lisa_queue_delete

```
lisa_err_t lisa_queue_delete(lisa_queue_t *queue)
```

参数：

queue：消息队列指针

返回值：

LISA_OK：　成功

LISA_FAIL：　失败

说明：

销毁消息队列

lisa_queue_available

```
uint32_t lisa_queue_available(lisa_queue_t *queue)
```

参数：

queue：消息队列指针

返回值：

消息队列中剩余可用的消息的个数

说明：

查询消息队列中空闲消息的个数

lisa_queue_waiting

```
uint32_t lisa_queue_waiting(lisa_queue_t *queue)
```

参数：

queue：消息队列指针

返回值：

消息队列中已有消息的个数

说明：

查询消息队列中正在等待的消息的个数

lisa_queue_clear

```
lisa_err_t lisa_queue_clear(lisa_queue_t *queue)
```

参数：

queue：消息队列指针

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

复位消息队列，此函数会清空消息队列

3. Mutex

lisa_mutex_create

```
lisa_mutex_t *lisa_mutex_create(void)
```

参数：

void

返回值：

lisa_mutex_t*：互斥量的指针

成功： 返回互斥量指针

失败： 返回 NULL

说明：

创建互斥量

lisa_mutex_lock

```
lisa_err_t lisa_mutex_lock(lisa_mutex_t *mutex, int32_t block_time)
```

参数：

mutex： 互斥量指针

block_time：超时时间，单位为 ms

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

互斥量上锁，如果超时时间到了，还无法获得锁，那么返回 LISA_FAIL；在超时时间到达前获得锁，那么返回 LISA_OK；

lisa_mutex_unlock

```
lisa_err_t lisa_mutex_unlock(lisa_mutex_t *mutex)
```

参数：

mutex：　　互斥量指针

返回值：

LISA_OK：　成功

LISA_FAIL：　失败

说明：

互斥量解锁

lisa_mutex_delete

```
lisa_err_t lisa_mutex_delete(lisa_mutex_t *mutex)
```

参数：

mutex：　　互斥量指针

返回值：

LISA_OK：　成功

LISA_FAIL：　失败

说明：

销毁互斥量

4 .Semaphore

lisa_semaphore_create

```
lisa_semaphore_t *lisa_semaphore_create(uint32_t count)
```

参数：

count： 信号量最大个数

返回值：

lisa_semaphore_t *： 信号量指针

成功： 返回信号量指针

失败： 返回 NULL

说明：

创建信号量

lisa_semaphore_take

```
lisa_err_t lisa_semaphore_take(lisa_semaphore_t *sem, int32_t block_time)
```

参数：

sem： 信号量指针

block_time：超时时间

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

获取信号量，如果超时时间到了，还无法获得信号量，那么返回 LISA_FAIL；在超时时间到达前获得信号量，那么返回 LISA_OK；

lisa_semaphore_give

```
lisa_err_t lisa_semaphore_give(lisa_semaphore_t *sem)
```

入参	注释
mutex	信号量指针

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

释放信号量

lisa_semaphore_delete

```
lisa_err_t lisa_semaphore_delete(lisa_semaphore_t *sem)
```

参数：

mutex：　　信号量指针

返回值：

LISA_OK：　成功

LISA_FAIL：　失败

说明：

销毁信号量

lisa_semaphore_getcount

```
uint32_t lisa_semaphore_getcount(lisa_semaphore_t *sem)
```

参数：

mutex：　　信号量指针

返回值：

可用的信号量个数

说明：

获取可用的信号量个数

5. Timer

lisa_timer_create

```
lisa_timer_t *lisa_timer_create(lisa_timertype type,
                                lisa_timercb_t cb,
```

```
                                                                    void *arg,
                                                                    uint32_t
period_ms)
```

参数：

type： 定时器类型

```
typedef enum {
    OS_TIMER_ONCE = 0, // 单次定时器
    OS_TIMER_PERIODIC = 1, // 周期定时器
} lisa_timertype;
```

cb： 定时器回调函数

```
typedef void (*lisa_timercb_t)(void *arg)
```

arg： 定时器回调函数的参数

period_ms：定时器的周期，单位 ms

返回值：

lisa_timer_t *： 定时器指针

成功： 返回定时器指针

失败： 返回 NULL

说明：

创建软件定时器

lisa_timer_delete

```
lisa_err_t lisa_timer_delete(lisa_timer_t *timer)
```

参数：

timer： 定时器指针

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

销毁定时器

lisa_timer_start

```
lisa_err_t lisa_timer_start(lisa_timer_t *timer)
```

参数：

timer：　　定时器指针

返回值：

LISA_OK：　成功

LISA_FAIL：　失败

说明：

启动定时器

lisa_timer_stop

```
lisa_err_t lisa_timer_stop(lisa_timer_t *timer)
```

参数：

timer：　　定时器指针

返回值：

LISA_OK：　成功

LISA_FAIL：　失败

说明：

停止定时器

lisa_timer_change_period

```
lisa_err_t lisa_timer_change_period(lisa_timer_t *timer, uint32_t period_ms)
```

参数：

timer: 定时器指针

period: 定时器周期，单位 ms

返回值：

LISA_OK: 成功

LISA_FAIL: 失败

说明：

修改定时器周期

6 .Time

lisa_os_get_ticks

```
uint32_t lisa_os_get_ticks(void)
```

参数：

void

返回值：

返回从 OS 启动后到现在为止的时间间隔，单位为时间片，时间片由具体 OS 实现决定

说明：

获得 OS 启动到现在的时间间隔，单位为时间片

lisa_os_get_time

```
uint32_t lisa_os_get_time(void)
```

参数：

void

返回值：

返回从 OS 启动后到现在为止的时间，单位为秒

说明：

获得 OS 启动到现在的时间，单位为秒

lisa_rand32

```
uint32_t lisa_rand32(void)
```

参数：

void

返回值：

返回随机值（直接返回 os 的当前 tick 值）

说明：

获得 32 位随机值

7.Memory

lisa_mem_alloc

```
void *lisa_mem_alloc(uint32_t size);
```

参数：

size：大小为 byte

返回值：

void*

说明：

申请内存空间

lisa_mem_realloc

```
void *lisa_mem_realloc(void *ptr, uint32_t size)
```

参数：

ptr：内存空间地址

size：申请内存大小，单位为 byte

返回值：

void*

成功：返回非 NULL 指针

失败：返回 NULL

说明：

重新申请内存空间

lisa_mem_calloc

```
void *lisa_mem_calloc(uint32_t count, uint32_t size)
```

参数：

count：内存块个数

size：单个内存块大小，单位为 byte

返回值：

void*

成功：返回非 NULL 指针

失败：返回 NULL

说明：

申请 count*size 的内存空间

lisa_mem_free

```
void lisa_mem_free(void *ptr)
```

参数：

ptr：内存空间指针

返回值：

void

说明：

释放内存空间

lisa_strdup

```
char *lisa_strdup(const char *s)
```

参数：

s：字符串

返回值：

char*

成功：返回非 NULL 指针

失败：返回 NULL

说明：

字符串拷贝函数

Modules

1 .button 检测

1.1 简介

基于 multibutton 封装按键接口,支持短按，长按，双击。

1.2 接口说明

```
lisa_err_t lisa_keys_init(lisa_keys_on_event event_cb)
```

参数

event_cb　　注册按键回调函数

```
#define KEY_TYPE_SHORT (1 << 0)                    //短按
#define KEY_TYPE_LONG (1 << 1)                     //长按
#define KEY_TYPE_DOUBLETICK (1 << 2)               //双击

typedef struct {
    uint8_t type_e;                                //按键事件类型
    uint8_t key_index;                             //按键号
} lisa_key_event;

typedef void (*lisa_keys_on_event)(lisa_key_event *event);
```

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

通过该函数注册按键事件回调函数，并且启动按键检测。另外还需结合硬件电路配置合适的 button_config g_button[]

下面配置对应硬件关联为：

按键 2,3 一起通过 adc6 检测，按键 2 单独按下电压检测值为 1.3 左右，按键 3 单独按下电压检测值为 0.9 左右；

按键 4,5 一起通过 adc7 检测，按键 4 单独按下电压检测值为 1.3 左右，按键 5 单独按下电压检测值为 0.9 左右；

```
button_config g_button[4] = {
        [0] = {.type = KEY_TYPE_SHORT | KEY_TYPE_LONG, //支持短按和长按
                .key_index = 2, //按键号
                .ad_confg.channel = 6, //通过通道 6 的 adc 检测
                .ad_confg.lowValue = 1100, //1100mv~1600mv 是按键按下状态。
                .ad_confg.highValue = 1600},

        [1] = {.type = KEY_TYPE_SHORT | KEY_TYPE_LONG,
                .key_index = 3,
                .ad_confg.channel = 6,
                .ad_confg.lowValue = 700,
                .ad_confg.highValue = 1100},

        [2] = {.type = KEY_TYPE_SHORT | KEY_TYPE_LONG,
                .key_index = 4,
                .ad_confg.channel = 7,
                .ad_confg.lowValue = 700,
                .ad_confg.highValue = 1100},
        [3] = {.type = KEY_TYPE_SHORT | KEY_TYPE_LONG,
                .key_index = 5,
                .ad_confg.channel = 7,
                .ad_confg.lowValue = 1100,
                .ad_confg.highValue = 1600},

};
```

1.3 使用示例

需结合硬件电路设计，配置合适的 button_config

```
void
```

```
lisa_button_cb(lisa_key_event *event)

{

        LISA_LOGE(TAG, "lisa_button_cb key %d    type %d", event->key_index,
event->type_e);

}



static int

lisa_button(int argc, char *argv[])

{

        LISA_LOGE(TAG, "lisa_button test //////////////////");

        lisa_keys_init(lisa_button_cb);



        return 0;

}



MSH_CMD_EXPORT(lisa_button, button testself);
```

2. led 显示

2.1 简介

通过 3 路 PWM 控制 RGB led 完成显示功能，支持常亮，关闭，闪烁，呼吸灯。

2.2 接口说明

```
lisa_err_t lisa_led_init(void)
```

参数

void

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

完成 led 显示模块初始化，配置相关 pwm 号

```
lisa_err_t lisa_rgbled_ctr(lisa_led_mode_e mode, uint32_t rgb, uint16_t flash_cnt)
```

参数

mode 显示模式

```
typedef enum {
    LED_MODE_ON = 0, //常亮
    LED_MODE_OFF,              //常灭
    LED_MODE_BREATH,           //呼吸灯
    LED_MODE_FLASHING,         //闪烁

    LED_MODE_INVALID,
} lisa_led_mode_e;
```

rgb 显示颜色 RGB_BLUE/RGB_GREEN/RGB_RED

flash_cnt 闪烁次数（闪烁模式下有效）

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

按参数进行灯效显示

2.3 使用示例

```
static int
lisa_led(int argc, char *argv[])
{
lisa_led_mode_e mode;
```

```
uint32_t rgb;
uint16_t cnt = 0;

if (argc < 2) {
LISA_LOGE(TAG, "usage: %s breath|on|off|flash RGB cnt\r\n", argv[0]);
return 0;
}

lisa_led_init();

if (strcmp(argv[1], "breath") == 0) {
rgb = atoi(argv[2]);
mode = LED_MODE_BREATH;
LISA_LOGE(TAG, "breath 0x%x\r\n", rgb);

} else if (strcmp(argv[1], "on") == 0) {
rgb = atoi(argv[2]);
mode = LED_MODE_ON;
LISA_LOGE(TAG, "on 0x%x\r\n", rgb);
} else if (strcmp(argv[1], "off") == 0) {
rgb = atoi(argv[2]);
mode = LED_MODE_OFF;
LISA_LOGE(TAG, "off 0x%x\r\n", rgb);
} else if (strcmp(argv[1], "flash") == 0) {
rgb = atoi(argv[2]);
cnt = atoi(argv[3]);
mode = LED_MODE_FLASHING;
LISA_LOGE(TAG, "flash 0x%x %d\r\n", rgb, cnt);
}

switch (rgb) {
case 0:
lisa_rgbled_ctr(mode, 0, cnt);
break;
case 1:
lisa_rgbled_ctr(mode, RGB_BLUE, cnt);
break;
case 2:
lisa_rgbled_ctr(mode, RGB_GREEN, cnt);
break;
case 3:
lisa_rgbled_ctr(mode, RGB_GREEN | RGB_BLUE, cnt);
break;
case 4:
```

```
lisa_rgbled_ctr(mode, RGB_RED, cnt);
break;
case 5:
lisa_rgbled_ctr(mode, RGB_RED | RGB_BLUE, cnt);
break;
case 6:
lisa_rgbled_ctr(mode, RGB_RED | RGB_GREEN, cnt);
break;
case 7:
lisa_rgbled_ctr(mode, RGB_RED | RGB_GREEN | RGB_BLUE, cnt);
break;
default:
break;
}

return 0;
}

MSH_CMD_EXPORT(lisa_led, led testself);
```

3 持久化

3.1 简介

持久化依赖于 FLASH 模块，主要用于保存设备的 deviceID 和用户的设置信息。 建议移植 EasyFlash 库，该 库可提供 key-value 持久化接口。且默认开启均衡磨损和掉电保护。

3.2 接口说明

```
lisa_err_t lisa_perst_init()
```

参数

void

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

持久化功能初始化

```
lisa_err_t lisa_perst_put_string(const char *k, const char *v)
```

参数

k　　key 值

v　　value

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

用于保存字符型型 key-value

```
lisa_err_t lisa_perst_get_string(const char *k, char **v)
```

参数

k　　key 值

v　　value

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

用于获取字符型型 key-value

```
lisa_err_t lisa_perst_put_int(const char *k, int v)
```

参数

k　　key 值

v　　value

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

用于保存 int 型 key-value

```
lisa_err_t lisa_perst_get_int(const char *k, int *out_v)
```

参数

k　　key 值

v　　value

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

用于获取 int 型 key-value

```
lisa_err_t lisa_perst_put_bool(const char *k, bool v)
```

参数

k　　key 值

v　　value

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

用于保存 bool 型 key-value

```
lisa_err_t lisa_perst_get_bool(const char *k, bool *out_v)
```

参数

k　　key 值

v　　value

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

用于获取 bool 型 key-value

```
lisa_err_t lisa_perst_delete(const char *k)
```

参数

k　　key 值

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

用于从数据中删除对应 key-value

```
lisa_err_t lisa_perst_clear(void)
```

参数

void

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

清除为默认状态

3.3 使用示例

```
static int
lisa_perst(int argc, char *argv[])
{
        int8_t useok = 0;
        if (argc < 2) {
                LISA_LOGE(TAG, "usage: %s set|get|del|clr|info string|int|bool key value \r\n", argv[0]);
                return 0;
        }

        if (strcmp(argv[1], "set") == 0) {
                if (strcmp(argv[2], "string") == 0) {
```

```
            char *key = NULL;
            char *value = NULL;
            key = argv[3];
            value = argv[4];
            LISA_LOGE(TAG, "set string %s %s \r\n", key, value);

            lisa_perst_init();
            lisa_perst_put_string(key, value);

            useok = 1;

    } else if (strcmp(argv[2], "int") == 0) {
            char *key = NULL;
            int value;
            key = argv[3];
            value = atoi(argv[4]);
            LISA_LOGE(TAG, "set int %s %d \r\n", key, value);

            lisa_perst_init();
            lisa_perst_put_int(key, value);

            useok = 1;
    } else if (strcmp(argv[2], "bool") == 0) {
            char *key = NULL;
            bool value;
            key = argv[3];
```

```
                value = atoi(argv[4]);
                LISA_LOGE(TAG, "set bool %s %d \r\n", key, value);

                lisa_perst_init();
                lisa_perst_put_bool(key, value);

                useok = 1;
            }
        } else if (strcmp(argv[1], "get") == 0) {
            if (strcmp(argv[2], "string") == 0) {
                char *key = NULL;
                char *value = NULL;
                key = argv[3];

                lisa_perst_init();
                lisa_perst_get_string(key, &value);

                LISA_LOGE(TAG, "get string %s %s \r\n", key, value);

                useok = 1;

            } else if (strcmp(argv[2], "int") == 0) {
                char *key = NULL;
                int value;
                key = argv[3];
```

```
                lisa_perst_init();
                lisa_perst_get_int(key, &value);

                LISA_LOGE(TAG, "get int %s %d \r\n", key, value);

                useok = 1;
        } else if (strcmp(argv[2], "bool") == 0) {
                char *key = NULL;
                bool value;
                key = argv[3];

                lisa_perst_init();
                lisa_perst_get_bool(key, &value);

                LISA_LOGE(TAG, "get bool %s %d \r\n", key, value);

                useok = 1;
        }
    } else if (strcmp(argv[1], "del") == 0) {
        char *key = NULL;
        char *value = NULL;
        key = argv[3];
        value = argv[4];

        lisa_perst_init();
        lisa_perst_delete(key);
```

```
                LISA_LOGE(TAG, "del string %s\r\n", key);


                useok = 1;
        } else if (strcmp(argv[1], "clr") == 0) {
                lisa_perst_init();


                lisa_perst_clear();


        } else if (strcmp(argv[1], "info") == 0) {
                lisa_perst_init();
                ef_print_env();
        }


        if (!useok) {
                LISA_LOGE(TAG, "usage: %s set|get|del string|int|bool key value \r\n",
argv[0]);
        }
}


MSH_CMD_EXPORT(lisa_perst, persistence testself);
```

4 播放器

4.1 简介

移植 LISA 系统需要当前环境支持双播放器实例，播放器 1 主要用于播放音乐和订阅的内容且，播放器二主 要用于播放 tts 提示音。

4.2 接口说明

```
lisa_player_t *lisa_player_init(lisa_player_params_t *params);
```

参数

params 播放器配置参数

```
typedef enum {
    LISA_PLAYER_TYPE_ONE = 0,
    LISA_PLAYER_TYPE_TWO,                    //只支持 mp3 和 wav 格式
    LISA_PLAYER_TYPE_MAX
} lisa_player_type_e;

typedef enum {
    LISA_PLAYER_EVENT_ERROR,
    LISA_PLAYER_EVENT_PREPARED,      //未支持
    LISA_PLAYER_EVENT_STARTED,       //已播放
    LISA_PLAYER_EVENT_STOPPED,       //停止
    LISA_PLAYER_EVENT_PAUSED,        //暂定
    LISA_PLAYER_EVENT_RESUMED,       //恢复播放
    LISA_PLAYER_EVENT_FINISHED,      //播放完毕
} lisa_player_event_e;

typedef struct {
    void *user;                                  //回调参数
    lisa_player_type_e type;         //播放器类型
    lisa_player_cb cb;                           //注册回调，回调事件见
lisa_player_event_e
```

```
} lisa_player_params_t;
```

返回值

成功返回播放器实例；失败返回 NULL

说明

用于创建播放器实例，注册对应播放器事件回调函数。注意播放器 LISA_PLAYER_TYPE_TWO 只支持 mp3 和 wav 格式

```
lisa_err_t lisa_player_prepare_url(lisa_player_t *ins, const char *url)
```

参数

ins 播放器实例

url URL 指针

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

播放前加载资源，可以是网络 url，也可以是打包在文件系统的资源路径

```
lisa_err_t lisa_player_set_callback(lisa_player_t *ins, lisa_player_cb callback)
```

参数

ins 播放器实例

callback 回调函数

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

播放器事件回调函数可以在 lisa_player_init 创建实例中注册，也可以通过该函数重新注册

```
lisa_err_t lisa_player_play(lisa_player_t *ins)
```

参数

ins 播放器实例

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

用于加载资源后，开始播放

```
lisa_err_t lisa_player_stop(lisa_player_t *ins)
```

参数

ins 播放器实例

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

停止播放

```
lisa_err_t lisa_player_pause(lisa_player_t *ins)
```

参数

ins 播放器实例

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

暂停播放

```
lisa_err_t lisa_player_resume(lisa_player_t *ins)
```

参数

ins 播放器实例

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

恢复播放

```
lisa_player_state_e lisa_player_get_state(lisa_player_t *ins)
```

参数

ins 播放器实例

返回值

当前播放器状态

```
typedef enum {
    LISA_PLAYER_STAUS_PLAYING = 0,          //正在播放
    LISA_PLAYER_STATUS_PAUSED,              //暂停播放
    LISA_PLAYER_STATUS_STOPPED,             //停止播放

    LISA_PLAYER_ERR,
} lisa_player_state_e;
```

说明

主动获取当前播放器状态

```
lisa_err_t lisa_player_seek(lisa_player_t *ins, uint32_t progress_ms)
```

参数

ins 播放器实例

progress_ms 待 seek 的时间点

返回值

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明

在播放过程中 seek music

```
uint32_t lisa_player_get_progress(lisa_player_t *ins)
```

参数

ins 播放器实例

返回值

当前播放进度时间，单位 ms

说明

在播放过程中获取当前播放器进度

```
uint32_t lisa_player_get_duration(lisa_player_t *ins)
```

参数

ins　播放器实例

返回值

当前歌曲总长，单位 ms

说明

在播放过程中获取当前歌曲总长

```
lisa_err_t lisa_player_set_volume(lisa_player_t *ins, uint8_t volume)
```

参数

ins　播放器实例

volume　音量大小，0~100

返回值

成功返回　LISA_OK;　失败返回　LISA_FAIL

说明

调整对应播放器音量大小

```
uint8_t lisa_player_get_volume(lisa_player_t *ins)
```

参数

ins　播放器实例

返回值

音量大小

说明

获取对应播放器音量大小

4.3 使用示例

```
lisa_player_t *g_player1 = NULL;
lisa_player_t *g_player2 = NULL;

static struct optparse_long opts[] = {{"version", 'V', OPTPARSE_NONE}, /* 版本 */
                        {"help", 'h', OPTPARSE_NONE}, /* 帮助 */
                        {"start", 's', OPTPARSE_REQUIRED}, /* 播放 */
                        {"stop", 't', OPTPARSE_NONE}, /* 停止 */
                        {"pause", 'p', OPTPARSE_NONE}, /* 暂停 */
                        {"resume", 'r', OPTPARSE_NONE}, /* 恢复 */
                        {"seek", 'k', OPTPARSE_REQUIRED}, /* 移动 */
                        {"volume", 'v', OPTPARSE_REQUIRED}, /* 音量 */
                        {"dump", 'd', OPTPARSE_NONE}, /* 信息 */
                        {NULL, 0, OPTPARSE_NONE}};

static void

usage(void)
{
        rt_kprintf("usage: player [type][option] [target] ...\n\n");
        rt_kprintf("type options: 1|2\n");

        rt_kprintf("usage options:\n");
        rt_kprintf("    -V,         --version                       Print player version
message.\n");
        rt_kprintf("    -h,         --help                          Print defined help message.\n");
```

```
        rt_kprintf(
                                "    -s URI, --start=URI                    Play music with
URI(network links or local "
                                "files).\n");
        rt_kprintf("    -t,         --stop                          Stop playing music.\n");
        rt_kprintf("    -p,         --pause                         Pause the music.\n");
        rt_kprintf("    -r,         --resume                        Resume the music.\n");
        rt_kprintf("    -k sec, --seek=sec                  Seek the specified seconds to
play.\n");
        rt_kprintf("    -v lvl, --volume=lvl                Change the volume(0~99).\n");
        rt_kprintf("    -d,         --dump                          Dump play relevant
information.\n");
}

#if 1
static void
dump_status(uint8_t type)
{
        lisa_player_t *playobj;
        const char *state[] = {"PLAYING", "PAUSED", "STOPPED"};
        if (1 == type) {
                playobj = g_player1;
        } else {
                playobj = g_player2;
        }

        rt_kprintf("\nPlayer Dump Status:\n");
```

```
        rt_kprintf("status      - %s\n", state[lisa_player_get_state(playobj)]);
        // rt_kprintf("URI           - %s\n", (player_get_uri() != NULL) ? player_get_uri() :
"NULL");
        rt_kprintf("volume      - %d\n", lisa_player_get_volume(playobj));
        rt_kprintf("codec       - %s\n", audio_codec_tostring(audio_codec_get()));

        if ((int)lisa_player_get_state(playobj) != (int)LISA_PLAYER_STATUS_STOPPED) {
                int value;

                value = lisa_player_get_duration(playobj);
                rt_kprintf("duration - %02d:%02d\n", value / 60, value % 60);

                value = lisa_player_get_progress(playobj) / 1000;
                rt_kprintf("position - %02d:%02d\n", value / 60, value % 60);
        }
}
#endif
void
player1_test_cb(lisa_player_event_e event, void *user)
{
        rt_kprintf("player1 event %d\n", event);
}

void
player2_test_cb(lisa_player_event_e event, void *user)
{
```

```
        rt_kprintf("player2 event %d\n", event);
}

static int
player_test(int argc, char *argv[])
{
        if (argc == 1) {
                usage();
                return RT_EOK;
        }

        uint8_t type = atoi(argv[1]);

        if (strcmp(argv[2], "-s") == 0) {
                rt_kprintf("player%d start in\n", type);
                if (1 == type) {
                        if (NULL == g_player1) {
                                lisa_player_params_t para;
                                memset(&para, 0, sizeof(lisa_player_params_t));
                                para.type = LISA_PLAYER_TYPE_ONE;
                                g_player1 = lisa_player_init(&para);
                        }

                        lisa_player_set_callback(g_player1, player1_test_cb);

                        lisa_player_prepare_url(g_player1, argv[3]);
```

```
                lisa_player_play(g_player1);

            } else {
                if (NULL == g_player2) {
                    lisa_player_params_t para;
                    memset(&para, 0, sizeof(lisa_player_params_t));
                    para.type = LISA_PLAYER_TYPE_TWO;
                    g_player2 = lisa_player_init(&para);
                }

                lisa_player_set_callback(g_player2, player2_test_cb);

                lisa_player_prepare_url(g_player2, argv[3]);

                lisa_player_play(g_player2);
            }

            rt_kprintf("player%d start out\n", type);
        } else if (strcmp(argv[2], "-p") == 0) {
            rt_kprintf("player%d pause in\n", type);
            if (1 == type) {
                lisa_player_pause(g_player1);
            } else {
                lisa_player_pause(g_player2);
            }
```

```
            rt_kprintf("player%d pause out\n", type);
        } else if (strcmp(argv[2], "-r") == 0) {
            rt_kprintf("player%d resume in\n", type);
            if (1 == type) {
                lisa_player_resume(g_player1);
            } else {
                lisa_player_resume(g_player2);
            }
            rt_kprintf("player%d resume in\n", type);
        } else if (strcmp(argv[2], "-t") == 0) {
            rt_kprintf("player%d stop in\n", type);
            if (1 == type) {
                lisa_player_stop(g_player1);
            } else {
                lisa_player_stop(g_player2);
            }
            rt_kprintf("player%d stop in\n", type);
        } else if (strcmp(argv[2], "-v") == 0) {
            uint8_t volume = atoi(argv[3]);
            rt_kprintf("player%d set volume%d in\n", type, volume);
            if (1 == type) {
                lisa_player_set_volume(g_player1, volume);
            } else {
                lisa_player_set_volume(g_player2, volume);
            }
            rt_kprintf("player%d set volume%d in\n", type, volume);
```

```
        } else if (strcmp(argv[2], "-d") == 0) {
                dump_status(type);
        } else if (strcmp(argv[2], "-k") == 0) {
                uint32_t seek = atoi(argv[3]);
                if (1 == type) {
                        lisa_player_seek(g_player1, seek);
                } else {
                        lisa_player_seek(g_player2, seek);
                }
                rt_kprintf("player%d seek %d\n", type, seek);
        }

        return RT_EOK;
}

MSH_CMD_EXPORT(player_test, persistence testself);
```

5 电源模块

5.1 简介

重启电源

5.2 接口说明

```
void lisa_reboot(void)
```

参数

void

返回值

void

说明

重启设备

6 异常跟踪

6.1 简介

用于保存异常时环境，log 输出相关信息。异常类型如下：

```
#define LISA_EXCEPTION_TYPE_DABT (1) //data abort
#define LISA_EXCEPTION_TYPE_PABT (2) //prefech abort
#define LISA_EXCEPTION_TYPE_UDEF (3) //未定义指令
#define LISA_EXCEPTION_TYPE_ASSERT (9)
```

6.2 接口说明

```
uint8_t exception_get(void)
```

参数

void

返回值

获取上次重启原因，见简介异常类型

说明

用于开机时获取上次重启原因

6.3 使用示例

下面是某次异常的日志输出，相关信息分析

7 CSK

7.1 简介

移植 LISA 系统需要当前环境支持双播放器实例，播放器 1 主要用于播放音乐和订阅的内容且，播放器二主 要用于播放 tts 提示音。

- LISA 系统可通过 UART 接口和 CSK 芯片进行交互
- LISA 系统可以通过 I2S 接口获取 CSK 芯片的音频，并传递给云端进行播音

7.2 接口说明

7.2.1 CSK 基本控制

csk_init

```
lisa_err_t csk_init(lisa_csk_cb_t cb, lisa_csk_err_cb_t cb_err)
```

参数：

cb： csk 回调函数

```
typedef void (*lisa_csk_cb_t)(int cmd);
```

cb_err： csk 错误回调函数

```
typedef void (*lisa_csk_err_cb_t)(int cmd);
```

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

csk sdk 的初始化

csk_exit

```
lisa_err_t csk_exit(void)
```

参数：

void

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

关闭 csk 能力

csk_get_all_version

```
lisa_err_t csk_get_all_version(csk_version_t *version, csk_version_type_e type)
```

参数：

version：版本号数组

```
typedef char csk_version_t[3][32];
```

type：版本的类型

```
typedef enum {
    e_csk_all_version = 0x00,           // 所有的版本号（包含固件、cae 算法、esr 算法的版本号）
    e_csk_master_version = 0x01,// 固件的版本号结果存储在 version[0]
    e_csk_cae_version = 0x02,           // cae 算法的版本号，结果存储在 version[1]
    e_csk_esr_version = 0x03,           // esr 算法的版本号，结果存储在 version[2]
} csk_version_type_e;
```

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

获取 CSK 代码的版本号；阻塞函数，主动让出时间片

csk_is_ready

```
bool csk_is_ready(void)
```

参数：

void

返回值：

true: csk 芯片已就绪

false: csk 芯片还未就绪

说明：

查看 csk 芯片是否初始化完成，非阻塞函数。如果返回值为 false，不能进行后续的 csk 函数操作

csk_hard_reset

```
void csk_hard_reset(void)
```

参数：

void

返回值：

void

说明：

复位 csk 芯片

csk_soft_reset

```
void csk_soft_reset(void)
```

参数：

void

返回值：

void

说明：

csk 固件复位，关闭 csk 的功能，使 csk 处于空闲状态

7.2.2 CSK 算法

csk_asr_start

```
lisa_err_t csk_asr_start(lisa_csk_asr_config_t config,
                                                    lisa_csk_asr_result_cb_t cb,
                                                    lisa_csk_asr_err_cb_t
err_cb);
```

参数：

config：算法的配置

```
typedef enum {
    e_lisa_csk_mode_ivw_only = 0, //  仅运行离线唤醒引擎
    e_lisa_csk_mode_esr_only = 1, //  仅运行离线识别引擎
    e_lisa_csk_mode_one_shot = 2, //  运行一次离线唤醒+一次离线识别交互模式
    e_lisa_csk_mode_mult_shot = 3, //  运行一次离线唤醒+多次离线识别交互模式
} lisa_csk_mode_e;

// @描 述：离线引擎配置
typedef struct {
    lisa_csk_mode_e mode; //  离线引擎模式
    uint32_t time_out; //  超时
    uint32_t beam; //  波束
    uint32_t voice_id; //  播音 id
    bool loop; //  单次、循环
} lisa_csk_asr_config_t;
```

cb：识别结果回调函数

```
typedef void (*lisa_csk_asr_result_cb_t)(bool is_wakeup, int kid, char *result)
参数：
is_wakeup：是否唤醒
kid：  命令词编号
result：  表示 csk 芯片的识别结果
result 的示例见下面的 json 字符串：
        {
            "rlt": [{
                "istart": 79,
                "iresid": 0,
                "iduration": 22,
                "nfillerscore": 25115,
                "nkeywordscore": 45112,
                "ncm": 1415,
                "ncmThreshold": 613,
```

```
                "keyword": "xiao3 fei1 xiao3 fei1",
                "nDelayFrame": -1515870811
            }]
        }
```

err_cb：识别错误回调函数

```
// 错误类型
typedef enum {
    e_csk_asr_error = 0, // 识别错误
    e_csk_asr_timeout = 1, // 识别超时
} lisa_csk_asr_err_type_e;

typedef void (*lisa_csk_asr_err_cb_t)(lisa_csk_asr_err_type_e type, char *mesg)
参数：
type: 错误类型，见上文；一般为识别超时
mesg: 一般为 NULL
```

返回值：

LISA_OK： 成功

LISA_FAIL： 失败

说明：

启动 csk 芯片的算法，非阻塞函数

csk_enter_esr

```
lisa_err_t csk_enter_esr(void)
```

参数：

void

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

控制 csk 芯片算法从唤醒模式进入识别模式

csk_exit_esr

```
lisa_err_t csk_exit_esr(void)
```

参数：

void

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

控制 csk 芯片算法从识别模式退回到唤醒模式

7.2.3 CSK 录音

csk_record_config

```
lisa_err_t csk_record_config(lisa_csk_record_cb_t cb, lisa_csk_record_err_cb_t err_cb)
```

参数：

cb：录音回调函数

```
// i2s 传入 4ch,16khz,16bit 的音频
typedef struct {
short *ch1;            // ch1 的数据地址
short *ch2;                    // ch2 的数据地址
short *ch3;                    // ch3 的数据地址
short *ch4;                    // ch4 的数据地址
} lisa_csk_record_data_t;

// 录音回调函数
typedef void (*lisa_csk_record_cb_t)(lisa_csk_record_data_t *audio, int samples);
参数：
        audio: 4 路音频的指针数组
        samples：每一路采样点个数
```

err_cb：录音错误回调

```
typedef void (*lisa_csk_record_err_cb_t)(int type, char *mesg);
```

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

csk 录音回调配置

csk_record_set_i2s_ch

```
lisa_err_t csk_record_set_i2s_ch(lisa_csk_record_config_t config)
```

参数：

config：csk 通过 i2s 传给上位机的音频配置

```
typedef enum {
    e_lisa_csk_record_channel_mute = 0,          // 静音数据
    e_lisa_csk_record_channel_mic1 = 1,          // mic1 的数据
    e_lisa_csk_record_channel_mic2 = 2,          // mic2 的数据
    e_lisa_csk_record_channel_ref1 = 3,          // 参考音频 1 的数据
    e_lisa_csk_record_channel_ref2 = 4,          // 参考音频 2 的数据
    e_lisa_csk_record_channel_cae1 = 5,          // cae 输出第一路音频
    e_lisa_csk_record_channel_cae2 = 6,          // cae 输出第二路音频
    e_lisa_csk_record_channel_test = 7,          // 测试音频（目的：调 i2s 收发一致性）
    e_lisa_csk_record_channel_cloud_asr = 8      // 送给云端的音频数据
} lisa_csk_record_i2s_channel_e;

// channels_mask：录音通道映射器
// 比特 15-12：表示通道 1；比特 11- 8：表示通道 2；比特 7 - 4：表示通道 3；比特 3 - 0：表示通道 4；
```

```
    // 比如 0x1205 表示，通道 1 映射到 MIC1，通道 2 映射到 MIC2，通道 3 纯零输出、通道 4 映射到 CAE1

        typedef struct {

        unsigned short channels_mask;

    } lisa_csk_record_config_t;
```

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

设置 csk 芯片通过 i2s 传输给上位机的音频；注意：i2s 音频格式为 16KHz，16bit，4ch

csk_record_start

```
lisa_err_t csk_record_start(void)
```

参数：

void

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

启动上位机的 i2s 录音

csk_record_stop

```
lisa_err_t csk_record_stop(void)
```

参数：

void

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

停止上位机的 i2s 录音

csk_record_set_uac_ch

```
lisa_err_t csk_record_set_uac_ch(lisa_csk_record_uac_in_e ch)
```

参数：

ch：csk 芯片传给上位机的音频

```
typedef enum {
    e_lisa_csk_record_uac_in_mute = 0,  // 静音数据
    e_lisa_csk_record_uac_in_cae1 = 1,  // cae 输出第一路音频
    e_lisa_csk_record_uac_in_cae2 = 2,  // cae 输出第二路音频
    e_lisa_csk_record_uac_in_cae3 = 3,  // cae 输出第三路音频，送给云端
    e_lisa_csk_record_uac_in_test = 4,  // 测试音频（用于调试）
} lisa_csk_record_uac_in_e;
```

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

如果上位机支持 UAC 功能，那么可以通过此函数设置 csk 芯片通过 UAC 传给上位机的音频

7.2.4 CSK 播音

csk_player_play

```
lisa_err_t csk_player_play(lisa_csk_play_config_t config, lisa_csk_play_cb_t cb)
```

参数：

config：播音列表

```
typedef struct {
    uint16_t vid[4];  // 播音 id
} lisa_csk_play_config_t;
```

cb：播音回调函数

```
typedef enum {
    e_lisa_csk_play_started = 0,    // 开始播音
    e_lisa_csk_play_finished = 1,           // 播音完成
    e_lisa_csk_play_error = 2,      // 播音错误（参数错误）
    e_lisa_csk_play_cancel = 3      // 取消播音
} lisa_csk_play_status_e;

typedef void (*lisa_csk_play_cb_t)(lisa_csk_play_status_e status);
```

返回值：

成功返回 LISA_OK; 失败返回 LISA_FAIL

说明：

控制 csk 芯片进行播音

7.3 使用示例

1. 基本初始化：

   ```
   csk_init(csk_cb, csk_cb_err);                                    // csk sdk 的基本初始化
   ```

2. 录音配置：

   ```
   csk_record_config(record_cb, record_err_cb);          // 录音回调函数配置
   csk_record_set_i2s_ch(record_config);                    // 设置录音通道号
   ```

3. 启动 csk 的识别算法

   ```
   csk_asr_start(asr_config, asr_result_cb, asr_err_cb); // 启动 csk 芯片识别算法
   ```

4. 启动上位机的录音

```
    csk_record_start();                                                    //
启动上位机录音
```

```
// csk 基本控制

void csk_cb(int cmd)

{

    // 此函数暂未用到

}


void csk_cb_err(int cmd)

{

    // 此函数暂未用到

}



void record_cb(lisa_csk_record_data_t *audio, int samples)

{

    // audio 音频数组指针，samples：采样点个数

    // 把第 3 路 pcm 音频送给 app 去处理

    send_pcm_to_app(audio->ch3, samples);

}


void record_err_cb(int type, char *mesg)

{

    // type：错误类型   mesg: 错误信息

}
```

```
static lisa_csk_record_config_t record_config = {
                    .channels_mask = 0x1256,    // csk 芯片输出给上位机 mic1，mic2，cae1，
cae2 四路音频
};


lisa_csk_asr_config_t config = {
                    .mode = e_lisa_csk_mode_mult_shot,    // 多轮交互模式
                    .beam = 1,
          // 初始化波束指向
                    .loop = true,                                                    // 循环
模式
                    .time_out = 20,                                                  // 识别
超时时间
                    .voice_id = 0,                                                   // 在
csk 启动识别前不播放音频
};


void asr_cb(bool is_wakeup, int kid, char *result)
{
    // is_wakeup：是否是唤醒
    // kid：                命令词 id；                例如唤醒词 kid 一般为 0 或者 501
    // result：      csk 识别算法的识别结果，例如

        LISA_LOGI(TAG, "asr result: wakeup(%d) kid(%d), record:%s", is_wakeup, kid, result);
}


void asr_err_cb(lisa_csk_asr_err_type_e type, char *mesg)
{
```

```
    // type:  一般为超时，  mesg 一般为 NULL

        LISA_LOGI(TAG, "asr err: type(%d)", type);

}


void csk_demo(void argc, char* argv[])

{

    // csk 的初始化

    csk_init(csk_cb, csk_cb_err);                                    // 初始化上位机中与 csk
有关的 uart,i2s 线程

    lisa_thread_mdelay(50);                                                  // 需要等待
csk_init 执行完



    // csk 录音回调配置

    csk_record_config(record_cb, record_err_cb);// 录音回调配置



    // csk 录音通道配置

    csk_record_set_i2s_ch(record_config);                 // 上位机录音音频选择

    lisa_thread_mdelay(50);                                                  // 需要等待
csk_record_set_i2s_ch 执行完



    // 启动 csk 芯片识别算法

    csk_asr_start(asr_config, asr_result_cb, asr_err_cb);

    lisa_thread_mdelay(50);                                                  // 需要等待
csk_asr_start 执行完



    // 启动上位机的录音

    csk_record_start();
```

```
    //
}


MSH_CMD_EXPORT(csk_demo, csk_demo);
```

8. 日志

8.1 简介

提供通用的日志打印函数，具备常用的日志分级输出、tag 标记的能力

8.2 接口说明

日志等级

```
#define LOG_LEVEL_NONE (0) //
#define LOG_LEVEL_ERROR (1) // 错误
#define LOG_LEVEL_WARN (2) // 警告
#define LOG_LEVEL_INFO (3) // 信息
#define LOG_LEVEL_DEBUG (4) // 调试
```

日志打印函数

```
LISA_LOGE(tag, format, ...); // 输出错误日志
LISA_LOGW(tag, format, ...); // 输出警告日志
LISA_LOGI(tag, format, ...); // 输出信息日志
LISA_LOGD(tag, format, ...); // 输出调试日志
```

断言

```
#define LISA_OK (int32_t)(0)
#define LISA_FAIL (int32_t)(-1)
```

```
LISA_APP_ASSERT(exp, fmt, ...) // 线程断言
LISA_ISR_ASSERT(exp, fmt, ...) // 中断断言
```

9 Http 模块

9.1 简介

实现 https/http 模块的统一封装，提供 http 通信所需的通用接口

9.2 接口说明

接口

```
lisa_http_t *lisa_http_init(lisa_http_request_t *req);
```

参数

req http 配置参数

```
typedef enum {
LISA_HTTP_GET,
LISA_HTTP_POST,
LISA_HTTP_PUT,
LISA_HTTP_PATCH,
LISA_HTTP_DELETE,
} lisa_http_method_e; // http 请求类型

typedef struct {
const void *buf; // 传入的数据
int32_t len; // buf 的数据长度
void *user; // 用户自定义类型
} lisa_http_data_t; // http 回调函数的入参类型

typedef struct {
lisa_http_event_e what; // 事件类型
lisa_http_error_e error; // 错误值
const char *header; //http 的 header
uint16_t status_code; // http 的 code 值
} lisa_http_event_t;

typedef struct {
lisa_http_method_e method; // 请求类型，目前支持 post 和 get
```

```
const char *url;
const char *headers;
void *body;
int32_t body_len; // 不定长度传 -1
uint32_t timeout_ms; // 等待延时（默认 10ms，暂不支持修改）
void *user; // 回调函数需传回的数据
void (*on_event)(lisa_http_event_t *event); // 注册 http 事件通知时的回调
void (*on_data)(lisa_http_data_t *data); // 注册 http 有数据收到时的回调函数
} lisa_http_request_t;
```

返回值

成功返回 http 实例；失败返回 NULL

说明

用于创建 http 实例，注册对应 http 意外事件和接收到数据的回调函数。

接口

```
lisa_http_err_e lisa_http_perform(lisa_http_t *ins);
```

参数

ins　　http 实例

返回值

```
typedef enum {
  LISA_HTTP_OK = 0,                    // http 请求执行成功
  LISA_HTTP_COMMON_ERR,                //
  LISA_HTTP_PARAM_ERROR,          // 参数错误
} lisa_http_err_e;
```

说明

http 的请求，目前实现 get 和 post 请求。

接口

```
lisa_http_err_e lisa_http_cleanup(lisa_http_t *ins);
```

参数

ins　　http 实例

返回值

```
typedef enum {
    LISA_HTTP_OK = 0,                          // http 请求执行成功
    LISA_HTTP_COMMON_ERR,                      //
    LISA_HTTP_PARAM_ERROR,          // 参数错误
} lisa_http_err_e;
```

说明

删除 http 实例，并清理内部开销。

9.3 使用示例

配置 http 请求参数：>> lisa_http_test init http://www.rt-thread.com/service/echo RT-Thread is an open source IoT operating system from China!

启动 http 发送请求：>>lisa_http_test perform

删除 http 配置并关闭 http 端口：>>lisa_http_test cancel

```
#define HEADERS "Content-Type: application/octet-stream\r\n"


lisa_http_t *test_http = NULL;


void
get_http_data_cb(lisa_http_data_t *data)
{
        rt_kprintf("get_auth_data_cb ");
        if (data->buf) {
                rt_kprintf("%s", (char *)data->buf);
```

```
                rt_kprintf("\n");
        }
        if (data->user) {
                rt_kprintf("%p", (char *)data->user);
        }
}
int
lisa_http_test(int argc, char **argv)
{
        lisa_http_request_t http_param;

        if (rt_strcmp(argv[1], "init") == 0) {
                strcpy(http_param.url, argv[2]);
                strcpy(http_param.body, argv[3]);
                strcpy(http_param.headers, HEADERS);
                http_param.body_len = strlen(argv[3]);
                http_param.method = LISA_HTTP_POST;
                http_param.timeout_ms = 10;
                http_param.on_data = get_http_data_cb;
                http_param.on_event = NULL;
                http_param.user = NULL;
                test_http = lisa_http_init(&http_param);
        } else if (rt_strcmp(argv[1], "cancel") == 0) {
                lisa_http_cleanup(test_http);
        }
        if (rt_strcmp(argv[1], "perform") == 0) {
```

```
                lisa_http_perform(test_http);
        } else {
                rt_kprintf("params error\n");
        }
        return RT_EOK;
}

MSH_CMD_EXPORT(lisa_http_test, lisa_http_test testself);
```

10 Websocket

10.1 简介

实现 Websocket 通讯的 api 接口封装,用于 LISA 系统与云端的通信交互。

10.2 接口说明

接口

```
lisa_ws_t *lisa_ws_init(lisa_ws_request_t *req);
```

参数

```
typedef enum {
    LISA_WS_OK = 0,
    LISA_WS_COMMON_ERR,
} lisa_ws_err_e;


typedef enum {
    LISA_WS_ON_ERROR,
    LISA_WS_ON_HEADER,              // 多次发生
```

```
    LISA_WS_ON_CONNECTED,                    // 连接成功
    LISA_WS_ON_DISCONNECTED,          // 断开连接
} lisa_ws_event_e;


typedef struct {
    lisa_ws_event_e what;                // websocket 事件类型
    const char *header;                  // websocket 结构中的 header 属性
    void *user;
} lisa_ws_event_t;


typedef enum {
    LISA_WS_TEXT,                              // text 文本文件
    LISA_WS_BIN,                               // 二进制数据（主要用于语音传输和固件传输）
} lisa_ws_data_type_e;


typedef struct {
    lisa_ws_data_type_e type;
    const void *buf;                                                    // 收到的数据
    uint32_t len;                                                       // 收到的数据长度，最长为 2048
    void *user;                                                         // 用户带入的参数
} lisa_ws_data_t;


typedef struct {
    char *scheme;                          // "ws" "wss"
```

```
    char *host;                                                    // ws
时，为 80，wss 时为 443

    char *path;                                                    //

    uint32_t timeout_ms;                                           // 数据等待超
时,暂不支持更改（单位： ms）

    void *user;                                                    // 用
户携带的传入到回调函数的参数

    void (*on_event)(lisa_ws_event_t *event);   // 状态变化时的状态事件回调

    void (*on_data)(lisa_ws_data_t *data);      // 收到数据时的回调函数

} lisa_ws_request_t;
```

返回值

websocket 实体句柄

```
typedef struct {
        char scheme[16];        // "ws" "wss"
        char host[512];         //  web 的 host
        char path[512];         //  web 的 path
        uint16_t port;          //  websocket 连接的端口号
} url_info_t;


typedef struct {
        url_info_t u_info;          // 远端 url 数据
        rws_socket *socket;                 // rws 库对应的的 socket 结构体

        uint32_t timeout_ms;    //  数据回应超时值（ms）
        void *user;                                 // 携带的原始用户数据

        void (*inter_on_event)(lisa_ws_event_t *event);         // 收到数据是用户回调函
数
```

```
    void (*inter_on_data)(lisa_ws_data_t *data);                    // 事件通知的回调函数
} lisa_ws_t;
```

说明

注册 Websocekt 实例，成功后，该 Websocket 接口收到的数据会通过调用用户注册函数 on_data()，收到的网络数据将作为 on_data 的参数 lisa_ws_data_t 中的 buf 传入。

接口

```
lisa_ws_err_e lisa_ws_connect(lisa_ws_t *ins);
```

参数

websocket 实体

返回值

成功返回：LISA_WS_OK； 失败返回：LISA_WS_COMMON_ERR

说明

将配置好的 websocket 实体与服务端建立连接。

接口

```
lisa_ws_err_e lisa_ws_disconnect(lisa_ws_t *ins);
```

参数

websocket 实体

返回值

成功返回：LISA_WS_OK； 失败返回：LISA_WS_COMMON_ERR

说明

将配置好的 websocket 实体与服务端断开连接。

接口

```
lisa_ws_err_e lisa_ws_send_text(lisa_ws_t *ins, const char *text);
```

参数

ins： websocket 实体

text： 发送的字符串

返回值

成功返回：LISA_WS_OK； 失败返回：LISA_WS_COMMON_ERR

说明

向 ins 实体发送 tex t 字符串

接口

```
lisa_ws_err_e lisa_ws_send_binary(lisa_ws_t *ins, const void *buf, uint32_t len);
```

参数

ins： websocket 实体

buf： 发送的二进制数据

len: 数据长度

返回值

成功返回：LISA_WS_OK； 失败返回：LISA_WS_COMMON_ERR

说明

发送 len 长度的数据 buf

接口

```
lisa_ws_err_e lisa_ws_cleanup(lisa_ws_t *ins);
```

参数

websocket 实体

返回值

成功返回：LISA_WS_OK； 失败返回：LISA_WS_COMMON_ERR

说明

10.3 使用示例

注册并配置 websocket 实例：>> lisa_rws_test init wss echo.websocket.org /

连接到配置好的 websocket 实例：>>lisa_rws_test conn

通过 websocket 发送文本数据：>>lisa_rws_test test hello,listenAI

通过 websocket 发送二进制数据：>>lisa_rws_test bin

断开 websocket 连接：>>lisa_rws_test disc

删除 websocket 配置并关闭 websocket 端口：>>lisa_rws_test clean

```
void
test_ws_data_cb(lisa_ws_data_t *data)
{
WS_DBG(TAG, "test_ws_data_cb.");
if (data->buf) {
rt_kprintf("%s", (char *)data->buf);
rt_kprintf("\n");
}
if (data->type == LISA_WS_TEXT) {
WS_DBG(TAG, " LISA_WS_TEXT");
} else if (data->type == LISA_WS_BIN) {
WS_DBG(TAG, " LISA_WS_BIN");
}
}

uint8_t bin_data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
static int
lisa_rws_test(int argc, char *argv[])
{
lisa_ws_request_t lisa_ws_req;
if (rt_strcmp(argv[1], "init") == 0) {
if (rt_strcmp(argv[2], "wss") == 0) {
lisa_ws_req.scheme = "wss";
} else {
lisa_ws_req.scheme = "ws";
}
lisa_ws_req.host = argv[3];
lisa_ws_req.path = argv[4];
lisa_ws_req.timeout_ms = 0;
lisa_ws_req.on_data = test_ws_data_cb;
lisa_ws_req.on_event = NULL;
lisa_ws_req.user = NULL;
lisa_ws = lisa_ws_init(&lisa_ws_req);
} else if (rt_strcmp(argv[1], "conn") == 0) {
lisa_ws_connect(lisa_ws);
} else if (rt_strcmp(argv[1], "disc") == 0) {
lisa_ws_disconnect(lisa_ws);
```

```
} else if (rt_strcmp(argv[1], "clean") == 0) {
lisa_ws_cleanup(lisa_ws);
} else if (rt_strcmp(argv[1], "text") == 0) {
lisa_ws_send_text(lisa_ws, argv[2]);
} else if (rt_strcmp(argv[1], "bin") == 0) {
lisa_ws_send_binary(lisa_ws, bin_data, sizeof(bin_data));
}

return RT_EOK;
}
MSH_CMD_EXPORT(lisa_rws_test, lisa_rws_test text);
```