电子工艺实训

ADC

一、ADC原理

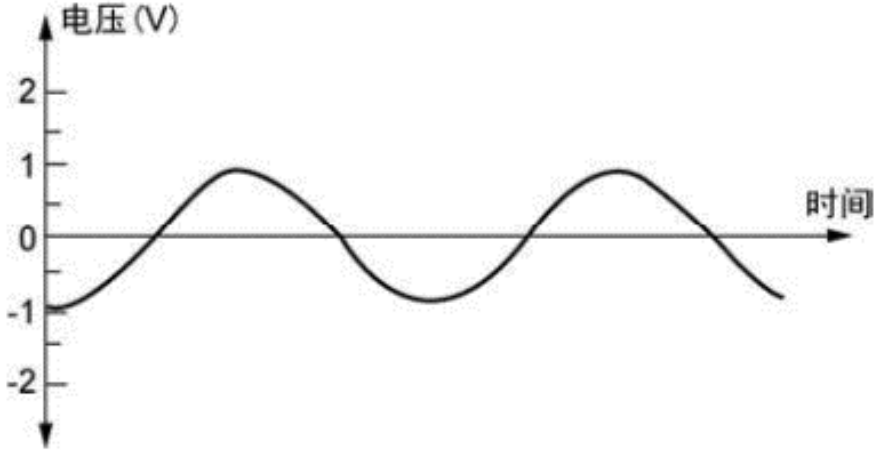
二、按键介绍

三、ADC案例程序：使用 ADC 值区分同一个引脚的不同按键

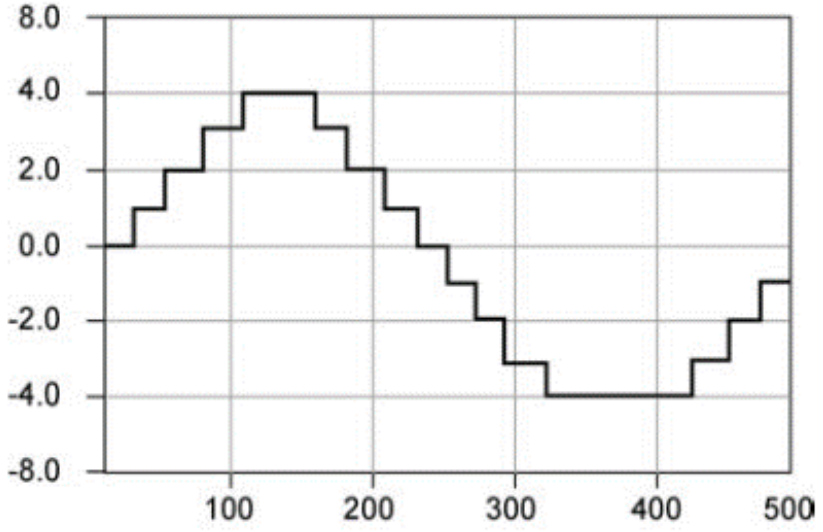
四、ADC实训练习：使用按键控制LED亮灭

一、ADC原理

1、模拟量与数字量



- **模拟量**
  - 连续地发生变化
  - 存储不方便



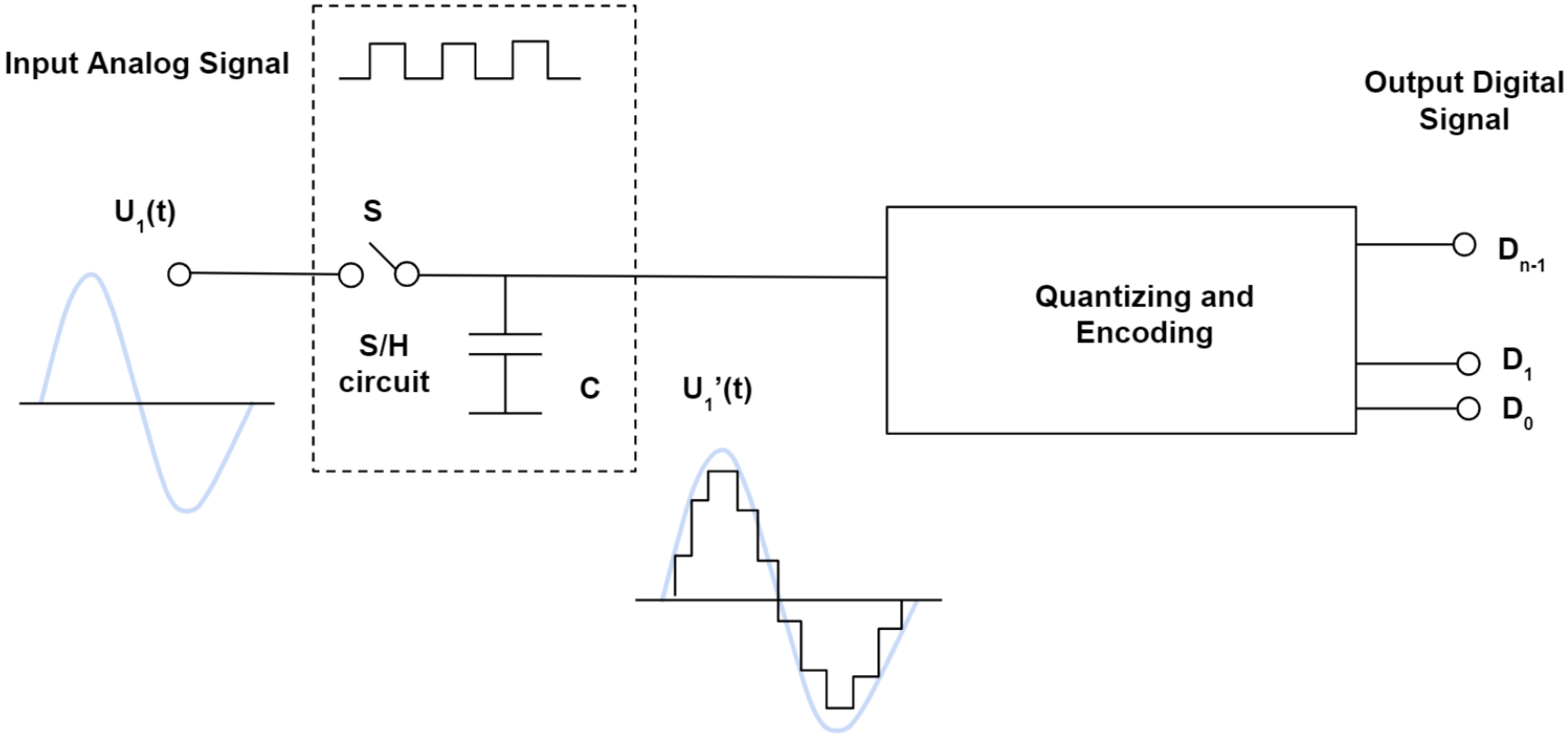
- **数字量**
  - 数据是分散开的
  - 方便存储

2、ADC（模数转换）原理

数字系统的最大特点是只能对输入的数字信号进行处理。在工业和生活中有很多物理量都是模拟量，比如温度、湿度、有害气体的含量等。这些模拟量可以通过相应的传感器变成与之对应的电压、电流等模拟信号。

为了实现数字系统对这些模拟信号的测量、运算和控制，就需要有一个从模拟信号到数字信号转换的过程，这个过程就叫模数转换。反之，将数字信号转换为模拟信号的过程，叫数模转换。

2、ADC（模数转换）原理


Input Analog Signal
Output Digital Signal
U1(t)
S
S/H circuit
C
U1'(t)
Quantizing and Encoding
Dn-1
D1
D0

3、Hi3861V100芯片 ADC 模块的功能特性

Hi3861V100 芯片的 ADC 模块具有以下功能特性：

- 输入时钟频率： 3MHz。
- 采样精度： 12bit。
- 单通道采样频率：小于 200kHz。
- 采样顺序：从通道 0 到通道 7，每个通道采样一个数据，循环采样。
- 支持采样数据平均滤波处理：平均次数 1、 2、 4、 8。在多通道场景下，每个通道接收 N 个数据（平均滤波个数）再切换通道。
- 模数转换电压基准：支持自动识别模式、 1.8V 基准模式、 3.3V 基准模式。

4、Hi3861V100芯片的 ADC 引脚分布

Hi3861V100 芯片有 8 个 ADC 通道，分别是 ADC0～ADC7。其中，通道 ADC7 是内部的 VBAT（ Voltage of the Battery，电池工作模式专用引脚）电压检测通道，不能进行 ADC 转换。通道 ADC0～ADC6 是 12 位逐次逼近型的 ADC 通道，用来实现将模拟信号转换为数字信号。

Hi3861V100 芯片的引脚数量是有限的，导致 ADC 功能没有独立的引脚，所以还是需要与 GPIO 进行引脚复用。

对于同时具有 GPIO 和 ADC 这两种功能（或更多功能）的引脚来说，同一时刻只能使用其中的一种功能。

4、Hi3861V100芯片的 ADC 引脚分布

引脚编号	默认功能	ADC 通道
6	GPIO-04	ADC1
17	GPIO-05	ADC2
19	GPIO-07	ADC3
27	GPIO-09	ADC4
29	GPIO-11	ADC5
30	GPIO-12	ADC0
31	GPIO-13	ADC6

5、相关 API 介绍

从 OpenHarmony1.0.1 版开始到 3.2 Beta4 版， HAL 接口缺失了 ADC 相关的 API。像下表所示的 API 已经不能再使用了。

API 名称	说明
unsigned int AdcRead(WifiIotAdcChannelIndex channel, unsigned short *data, WifiIotAdcEquModelSel equModel, WifiIotAdcCurBais curBais, unsigned short rstCnt);	读取 ADC 通道的值

5、相关 API 介绍

我们可以使用海思 SDK 的接口，接口位置如下：

device\hisilicon\hispark_pegasus\sdk_liteos\include\hi_adc.h

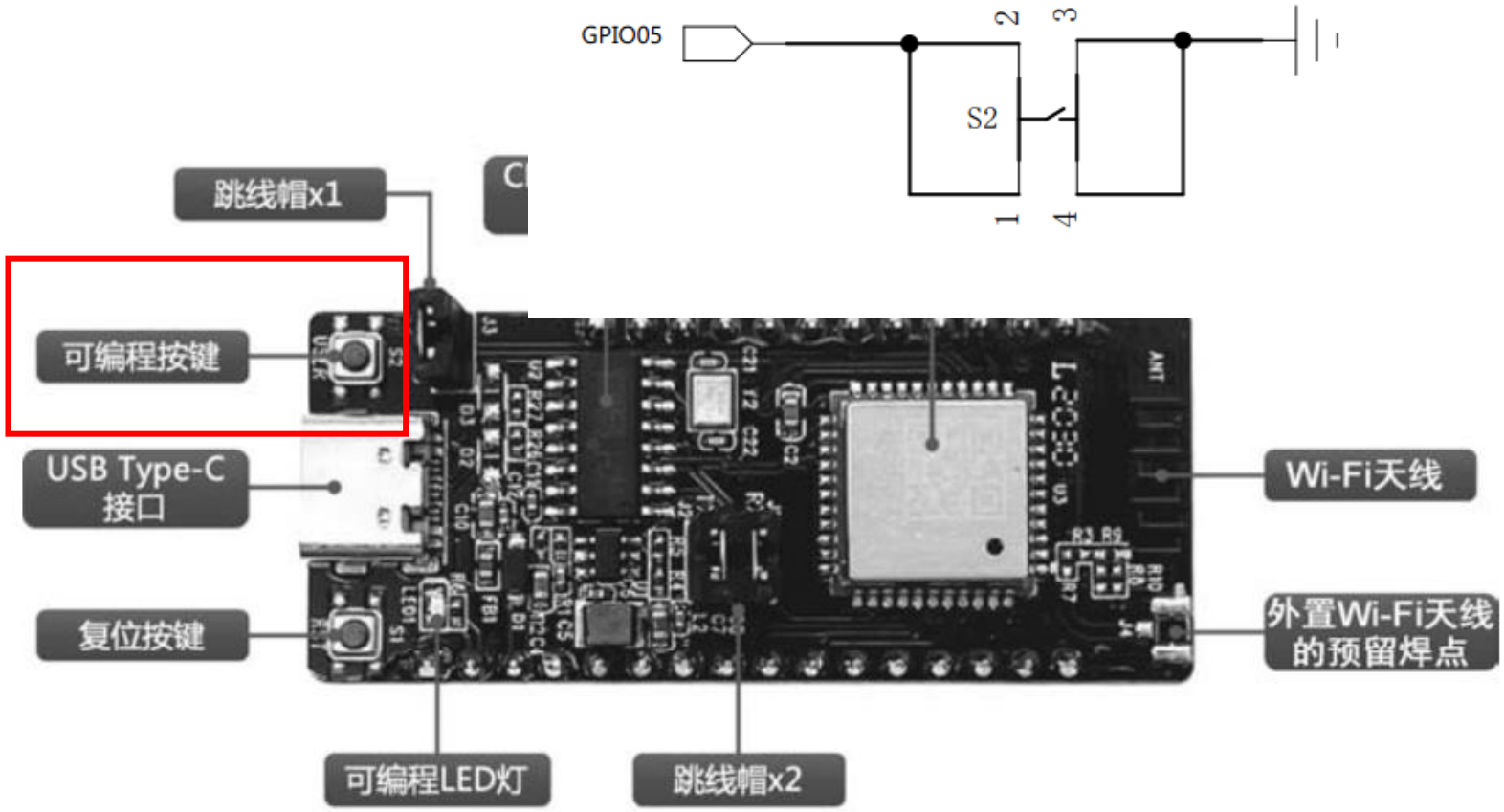
注意， ADC 读取数据的速度较慢，应当尽量避免在中断处理函数中使用。

目前学习这一个 ADC 相关 API 就可以了。

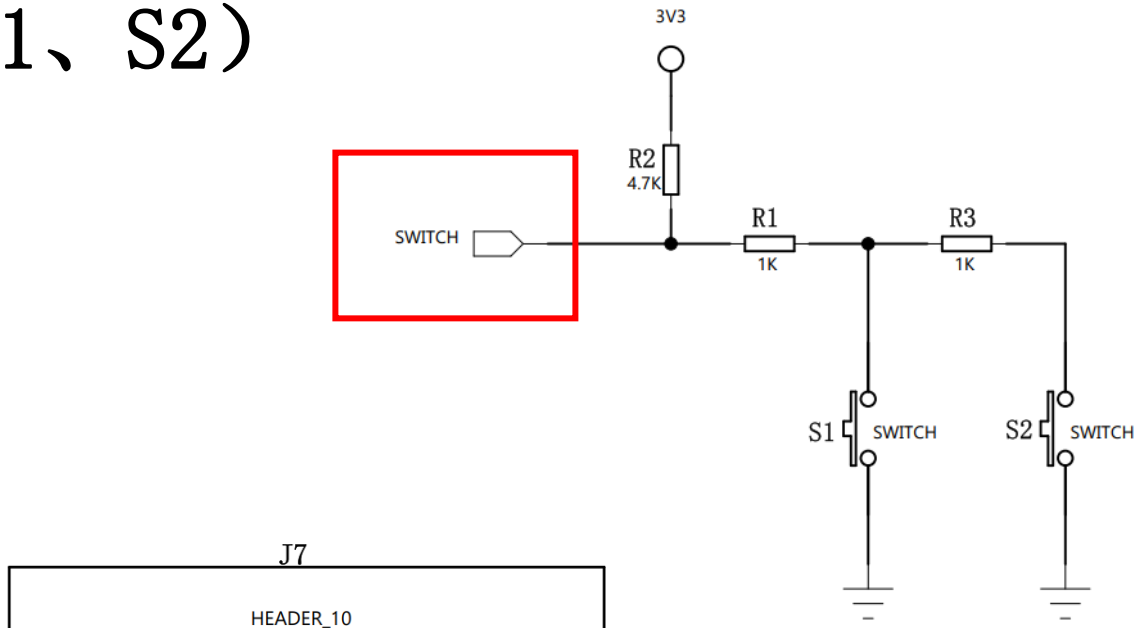
API 名称	说明
hi_u32 hi_adc_read(hi_adc_channel_index channel, hi_u16 *data, hi_adc_equ_model_sel equ_model, hi_adc_cur_bais cur_bais, hi_u16 delay_cnt)	从一个 ADC 通道中读一个数据。 参数 channel 用于指定 ADC 通道，参数 data 用于指定读取的 ADC 数据的保存地址，参数 equ_model 用于指定平均算法模式，参数 cur_bais 用于指定模拟电源控制（模数转换电压基准），参数 delay_cnt 用于指定从配置采样到启动采样的延时时间计数，一次计数时间是 334ns

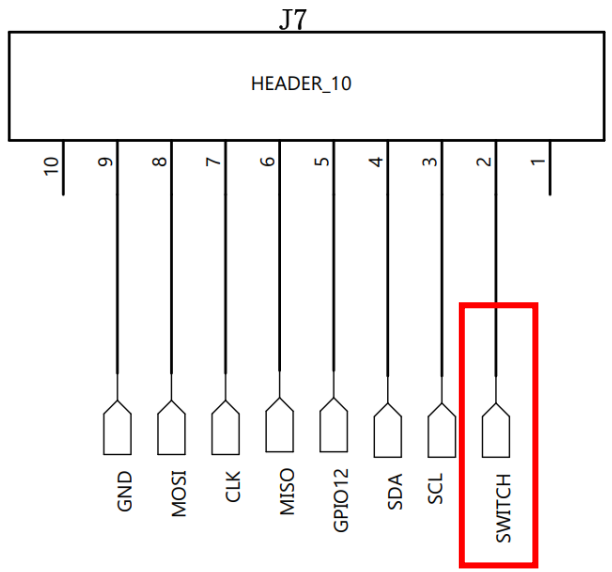
二、按键介绍

1、核心板按键（这里记作S3）

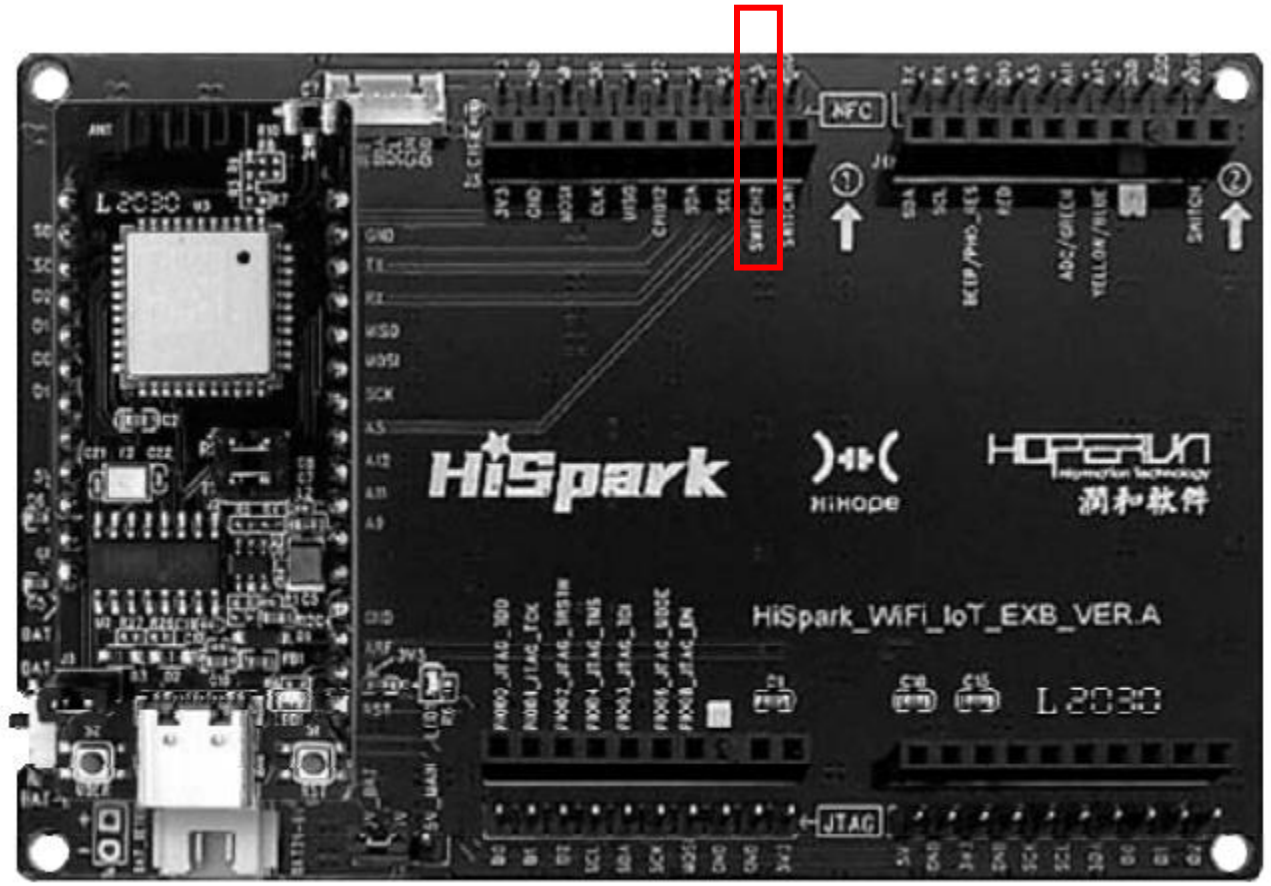


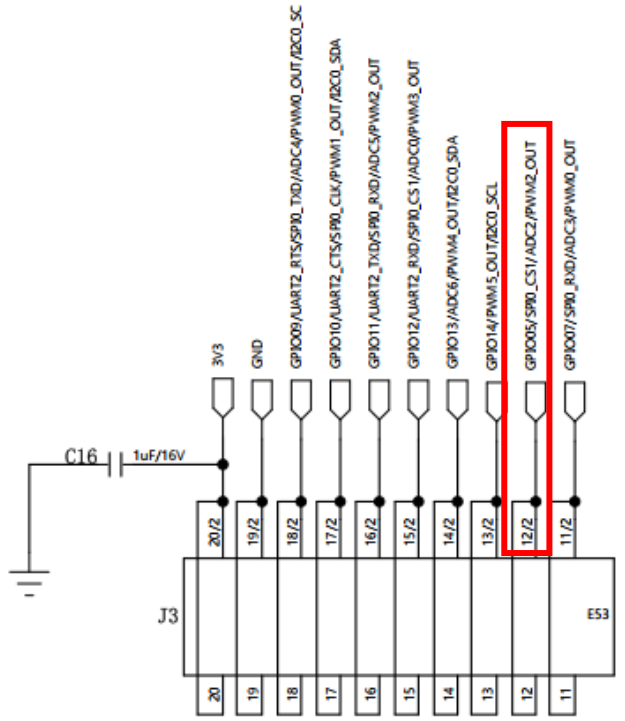
2、OLED 显示屏板的按键（S1、S2）





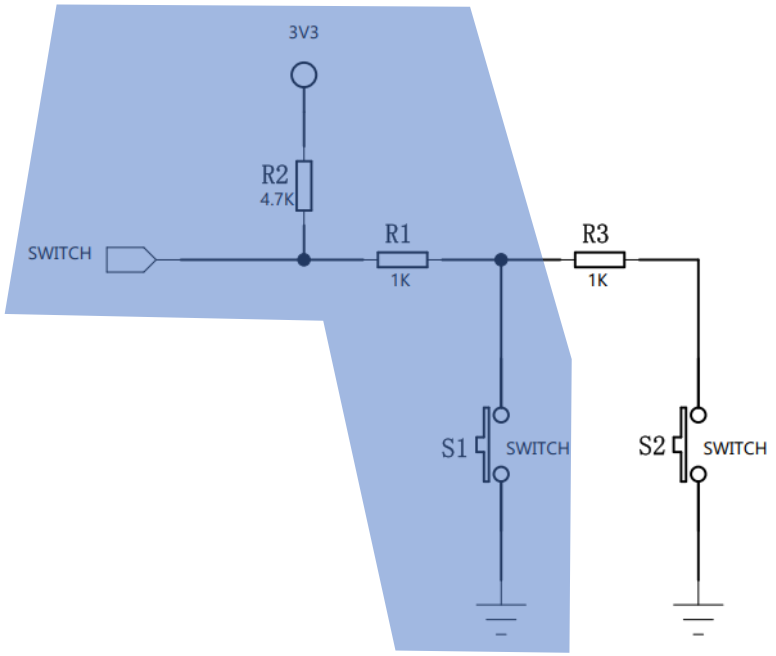
3、底板排插具体针脚位置





GPIO05/SPI0_CS1/ADC2/PWM2_OUT

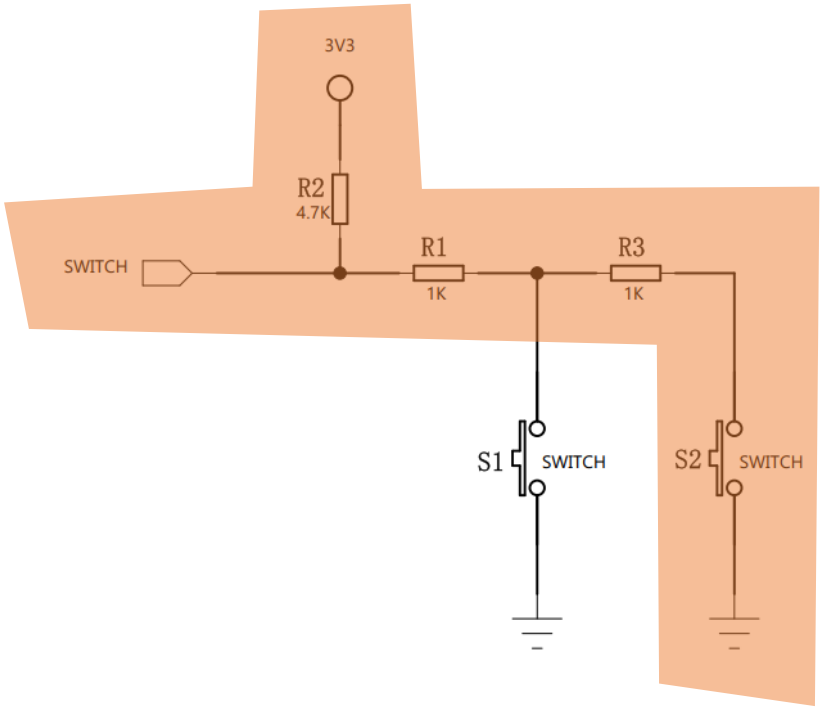
4、按键、电压与ADC值关系



按键 S1：

- 按键 S1连接到了 Hi3861V100 芯片的 17 号引脚上， 17号引脚的复用关系为 GPIO-05、PWM2 和 ADC2。
- 当按键 S1 被按下的时候， ADC 值在 228～455；
- 当按键 S1 抬起的时候， ADC 值在 1422～1820。
- 我们可以直接使用 ADC 值判断按键的状态，也可以将 ADC 值转换成电压再进行判断。

4、按键、电压与ADC值关系

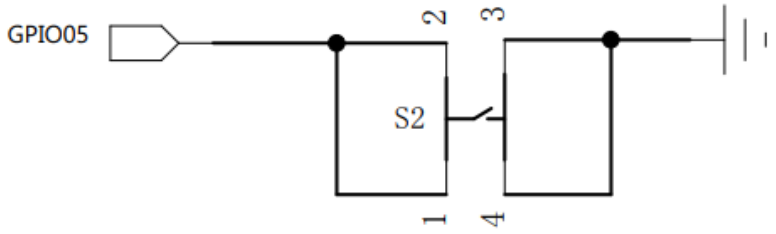


按键 S2：

- 按键 S2也是连接到芯片的 17 号引脚上。
- 当按键 S2 被按下的时候， ADC 值在 455～682；
- 当按键 S2 抬起的时候， ADC 值在 1422～1820。

按键 S3：

- 按键 S3也是连接到芯片的 17 号引脚上。
- 当按键 S3 被按下的时候， ADC 值在 5～228；
- 当按键 S3 抬起的时候， ADC 值在 1422～1820。



4、按键、电压与ADC值关系

按键描述	电压下限（V）	电压上限（V）	ADC 值下限	ADC 值上限
核心板的可编程按键 S3	0.01	0.4	5	228
OLED 显示屏板的 S1 按键	0.4	0.8	228	455
OLED 显示屏板的 S2 按键	0.8	1.2	455	682
无按键被按下	2.5	3.2	1422	1820

三、ADC案例程序：使用 ADC 值区分同一个引脚的不同按键

1、API接口

我们可以使用海思 SDK 的接口，接口位置如下：

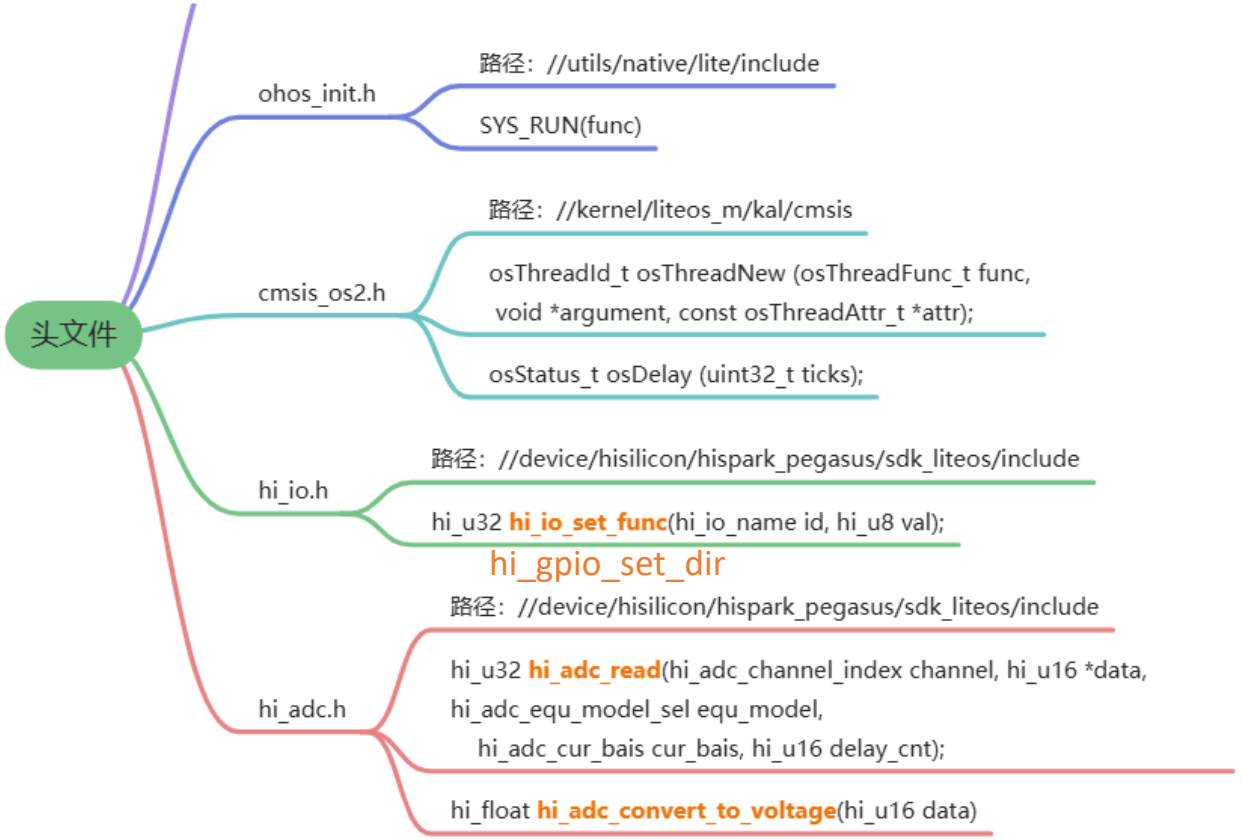
device\hisilicon\hispark_pegasus\sdk_liteos\include\hi_adc.h

注意， ADC 读取数据的速度较慢，应当尽量避免在中断处理函数中使用。

目前学习这一个 ADC 相关 API 就可以了。

API 名称	说明
hi_u32 hi_adc_read(hi_adc_channel_index channel, hi_u16 *data, hi_adc_equ_model_sel equ_model, hi_adc_cur_bais cur_bais, hi_u16 delay_cnt)	从一个 ADC 通道中读一个数据。 参数 channel 用于指定 ADC 通道，参数 data 用于指定读取的 ADC 数据的保存地址，参数 equ_model 用于指定平均算法模式，参数 cur_bais 用于指定模拟电源控制（模数转换电压基准），参数 delay_cnt 用于指定从配置采样到启动采样的延时时间计数，一次计数时间是 334ns

2、相关头文件


头文件
ohos_init.h
路径：//utils/native/lite/include
SYS_RUN(func)
cmsis_os2.h
路径：//kernel/liteos_m/kal/cmsis
osThreadId_t osThreadNew (osThreadFunc_t func,
void *argument, const osThreadAttr_t *attr);
osStatus_t osDelay (uint32_t ticks);
hi_io.h
路径：//device/hisilicon/hispark_pegasus/sdk_liteos/include
hi_u32 hi_io_set_func(hi_io_name id, hi_u8 val);
hi_gpio_set_dir
hi_adc.h
路径：//device/hisilicon/hispark_pegasus/sdk_liteos/include
hi_u32 hi_adc_read(hi_adc_channel_index channel, hi_u16 *data,
hi_adc_equ_model_sel equ_model,
hi_adc_cur_bais cur_bais, hi_u16 delay_cnt);
hi_float hi_adc_convert_to_voltage(hi_u16 data)

3、参考代码

我们在src\applications\sample\wifi-iot\app目录下新建adc_key项目目录，添加adc_key.c源文件和BUILD.gn配置文件。

包含的头文件作用如下：

```
#include <stdio.h>

#include <unistd.h>
#include <hi_types_base.h>
#include <hi_io.h>
#include <hi_early_debug.h>
#include <hi_gpio.h>
#include <hi_task.h>
#include "ohos_init.h"
#include "cmsis_os2.h"

#include <unistd.h>
#include <hi_types_base.h>

#define APP_DEMO_ADC

#include <hi_adc.h>
#include <hi_stdlib.h>
#include <hi_early_debug.h>
```

提供标准输入输出函数（如 printf），用于串口日志打印
提供 POSIX 系统调用（如 usleep），用于微秒级延时控制
定义 Hi3861 的基础数据类型（如 hi_u32, hi_u16），确保硬件兼容性
提供 GPIO 复用功能配置接口（如 hi_io_set_func），用于切换引脚功能模式（如 GPIO/ADC）
支持早期调试功能（如 hi_early_debug_init），用于系统启动前的日志输出
提供 GPIO 操作接口（如 hi_gpio_set_dir），配置引脚输入/输出方向及中断
任务管理 API（如 hi_task_create），已过时（被 CMSIS-RTOS2 替代）
定义 OpenHarmony 初始化宏（如 SYS_RUN），用于注册应用入口函数
CMSIS-RTOS2 多线程 API（如 osThreadNew），用于创建和管理 RTOS 任务
提供 ADC 操作接口（如 hi_adc_read），读取模拟通道电压值
扩展标准库函数（如 memset_s），提供安全内存操作

3、参考代码

adc_key.c源文件中的宏定义和变量声明：

```
#define KEY_EVENT_NONE      0
#define KEY_EVENT_S1        1
#define KEY_EVENT_S2        2
#define KEY_EVENT_S3        3
#define KEY_EVENT_S4        4

#define ADC_TEST_LENGTH  64
#define VLT_MIN 100
```

```
hi_u16 g_adc_buf[ADC_TEST_LENGTH] = { 0 };



int key_status = KEY_EVENT_NONE;
char key_flg = 0;
```

ADC_TEST_LENGTH 表示一个按键采样次数，多采样几次取平均值

3、参考代码

adc_key.c源文件中的get_key_event函数：

```
int get_key_event(void)
{
    int tmp = key_status;
    key_status = KEY_EVENT_NONE;
    return tmp;
}
```

int get_key_event(void) 获取当前按键状态

3、参考代码

adc_key.c源文件中的convert_to_voltage函数1：

```
/* asic adc test  */
hi_void convert_to_voltage(hi_u32 data_len)
{
    hi_u32 i;
    float vlt_max = 0;
    float vlt_min = VLT_MIN;

    float vlt_val = 0;

    hi_u16 vlt;
    for (i = 0; i < data_len; i++) {
        vlt = g_adc_buf[i];
        float voltage = (float)vlt * 1.8 * 4 / 4096.0;  /* vlt * 1.8 * 4 / 4096.0: Convert code int
        vlt_max = (voltage > vlt_max) ? voltage : vlt_max;
        vlt_min = (voltage < vlt_min) ? voltage : vlt_min;
    }
//    printf("vlt_min:%.3f, vlt_max:%.3f \n", vlt_min, vlt_max);
```

hi_void convert_to_voltage(hi_u32 data_len)

ADC值转换为电压值再进行判断，也可以不用这个函数直接通过ADC值判断

3、参考代码

adc_key.c源文件中的convert_to_voltage函数2：

```
vlt_val = (vlt_min + vlt_max)/2.0;

if((vlt_val > 0.4) && (vlt_val < 0.6))
{
    if(key_flg == 0)
    {
        key_flg = 1;
        key_status = KEY_EVENT_S1;
    }
}
if((vlt_val > 0.8) && (vlt_val < 1.1))
{
    if(key_flg == 0)
    {
        key_flg = 1;
        key_status = KEY_EVENT_S2;
    }
}
```

```
    if((vlt_val > 0.01) && (vlt_val < 0.3))
    {
        if(key_flg == 0)
        {
            key_flg = 1;
            key_status = KEY_EVENT_S3;
        }
    }

    if(vlt_val > 3.0)
    {
        key_flg = 0;
        key_status = KEY_EVENT_NONE;
    }
}
```

3、参考代码

adc_key.c源文件中的app_demo_adc_test函数：

```
void app_demo_adc_test(void)
{
    hi_u32 ret, i;
    hi_u16 data;  /* 10 */

    memset_s(g_adc_buf, sizeof(g_adc_buf), 0x0, sizeof(g_adc_buf));

    for (i = 0; i < ADC_TEST_LENGTH; i++) {
        ret = hi_adc_read((hi_adc_channel_index)HI_ADC_CHANNEL_2, &data, HI_ADC_EQU_MODEL_1, HI_ADC_CUR_BAIS_DEFAULT, 0);
        if (ret != HI_ERR_SUCCESS) {
            printf("ADC Read Fail\n");
            return;
        }
        g_adc_buf[i] = data;
    }
    convert_to_voltage(ADC_TEST_LENGTH);

}
```

void app_demo_adc_test(void)

ADC转换值依次存入g_adc_buf中，再调用convert_to_voltage转换为对应的电压值

3、参考代码

adc_key.c源文件中的my_gpio_adc_demo函数1：

```
void my_gpio_adc_demo(void *arg)
{
    arg = arg;
    hi_u32 ret;

    (hi_void)hi_gpio_init();

    hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO); /* uart1 rx */

    ret = hi_gpio_set_dir(HI_GPIO_IDX_5, HI_GPIO_DIR_IN);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR ======gpio -> hi_gpio_set_dir1 ret:%d\r\n", ret);
        return;
    }
```

void my_gpio_adc_demo(void *arg)

先用hi_io_set_func初始化GPIO_5作为GPIO与ADC复用功能，再用hi_gpio_set_dir设置引脚方向

3、参考代码

adc_key.c源文件中的my_gpio_isr_demo函数2：

```
while(1)
{
    //读取ADC值
    app_demo_adc_test();

    switch(get_key_event())
    {
        case KEY_EVENT_NONE:
        {

        }
        break;

        case KEY_EVENT_S1:
        {
            printf("KEY_EVENT_S1 \r\n");
        }
        break;
        case KEY_EVENT_S2:
        {
            printf("KEY_EVENT_S2 \r\n");
        }
        break;

        case KEY_EVENT_S3:
        {
            printf("KEY_EVENT_S3 \r\n");
        }
        break;

    }

    usleep(30000);
}

}
```

然后调用app_demo_adc_test得到按键电压，判断是哪个按键被按下去

3、参考代码

adc_key.c源文件中的key_demo函数：

```
void key_demo(void)
{
    osThreadAttr_t attr;

    attr.name = "KeyTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 2048;
    attr.priority = 26;

    if (osThreadNew((osThreadFunc_t)my_gpio_isr_demo, NULL, &attr) == NULL) {
        printf("[key_demo] Falied to create KeyTask!\n");
    }

}


SYS_RUN(key_demo);
```

void key_demo(void)
标准的线程生成osThreadNew，最后用SYS_RUN(key_demo);启动本例程

3、参考代码

adc_key目录中的BUILD.gn配置文件：

```
static_library("adc_key") {
    sources = [
        "adc_key.c"
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//kernel/liteos_m/components/cmsis/2.0",
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//device/soc/hisilicon/hi3861v100/hi3861_adapter/hals/communication/wifi_lite/wifiservice",
        "//device/soc/hisilicon/hi3861v100/hi3861_adapter/kal",
    ]
}
```

3、参考代码

上层app目录中的BUILD.gn配置文件：

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "adc_key:adc_key",

    ]
}
```

四、ADC实训练习：使用按键控制LED亮灭

- 按OLED板载的按键S1，打开主板的LED灯， 同时串口调试中同步更新显示LED灯状态LED ON/OFF
- 按OLED板载的按键S2，关闭主板的LED灯， 同时串口调试中同步更新显示LED灯状态LED ON/OFF
- 按主板的自定义按键S3，切换主板的LED灯状态， 同时串口调试中同步更新显示LED灯状态LED ON/OFF