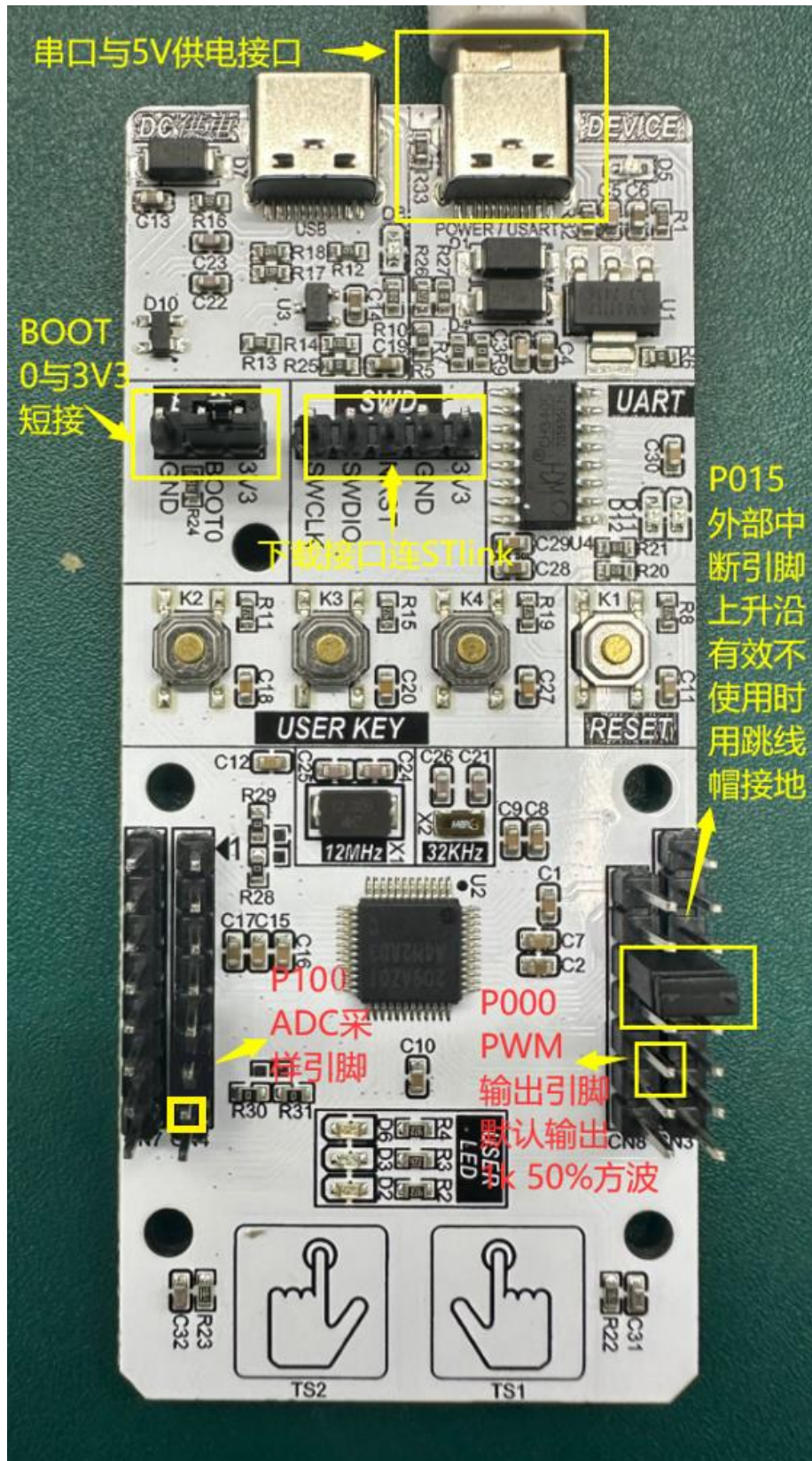


# 瑞萨 RA4M2 模板使用说明

## 目录

1 开发板引脚使用与接线.....	2
2 模板代码功能.....	4
2.1 LED 与按键功能.....	4
2.2 定时器功能.....	4
2.3 片内温度传感器功能.....	4
2.4 ADC 功能.....	4
2.5 串口功能.....	4
2.6 外部中断功能.....	5
3 代码介绍及函数实现.....	6
3.1 bsp_system.....	6
3.2 scheduler.....	7
3.3 hal_entry.....	8
3.4 key_app.....	9
3.5 led_app.....	11
3.6 uart_app.....	12
3.7 adc_app.....	13
3.8 tsn_app.....	14
3.9 tim_app.....	15
3.10 external_app.....	16

## 1 开发板引脚使用与接线



\*注：本文提到的板上器件都会以板上丝印标注为准 如按键 K2、K3、K3，LED D2、

D3、D6 等。

Cortex-M Target Driver Setup

Debug

Trace

Flash Download

Pack

Download Function

LOAD

☐ Erase Full Chip

☒ Program

☒ Erase Sectors

☒ Verify

☐ Do not Erase

☒ Reset and Run

RAM for Algorithm

Start: 0x20000000

Size: 0x00007800

Programming Algorithm

Description	Device Size	Device Type	Address Range
RA4M2 512K Flash	512k	On-chip Flash	00000000H - 0007FFFFH
RA4M2 8KB DataFlash	8k	On-chip Flash	08000000H - 08001FFFH
RA4M2 Config Area	512B	On-chip Flash	0100A100H - 0100A2FFFH

Start:

Size:

Add

Remove

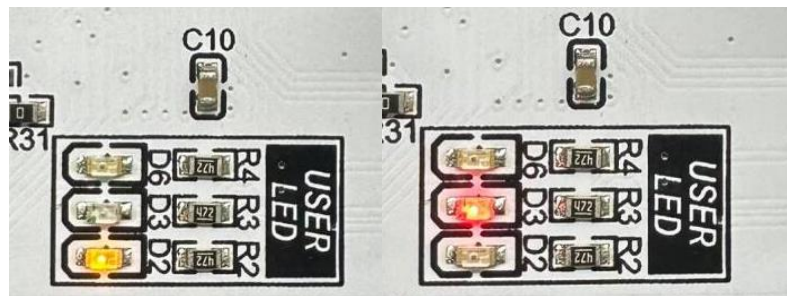
Flash Download 配置

\*注：开发板下载代码以后无论是否勾选“Reset and Run”都需要按下 K1 进行复位  
后代码才可以正常运行

## 2 模板代码功能

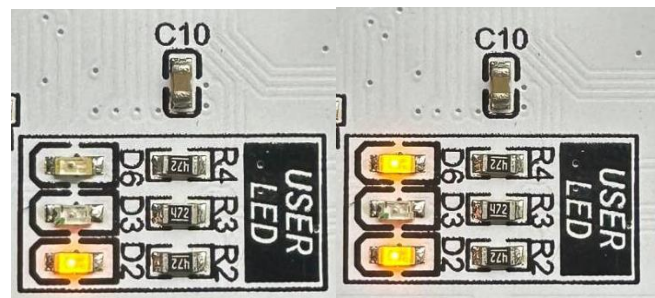
### 2.1 LED 与按键功能

当 K2 按下时，LED 会在 D2 与 D3 之间切换点亮



### 2.2 定时器功能

定时器设置为 1ms 定时器，当定时器计满 500ms 时 D6 取反，实现 LED 以 1s 为周期闪烁



### 2.3 片内温度传感器功能

读取 CPU 内部温度

### 2.4 ADC 功能

读取 P100 上电压

### 2.5 串口功能

上电时会发送：

```
/*system_init*/
```

之后每隔 2s 发送如下格式的数据：

```
*****BEGIN*****
```

```
CPU Temperature: 00.00
```

```
ADC data:0.00V
```

```
*****END*****
```

CPU 温度与 ADC 采样值均保留小数点后 2 位

## 2.6 外部中断功能

当 P015 接收到上升沿时发送:

P015 interrupt!

## 2.7 PWM 功能

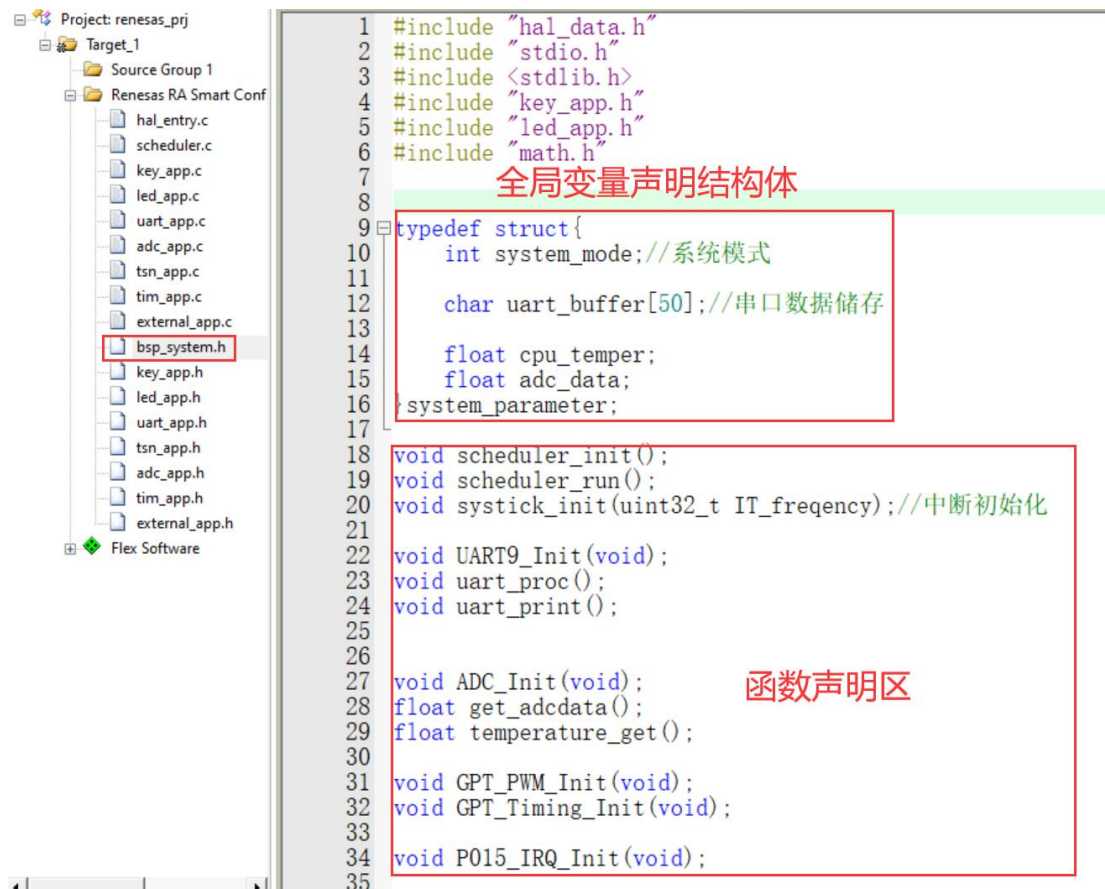
P000 上电后会输出 1K 50%占空比的正弦波





## 3 代码介绍及函数实现

### 3.1 bsp\_system

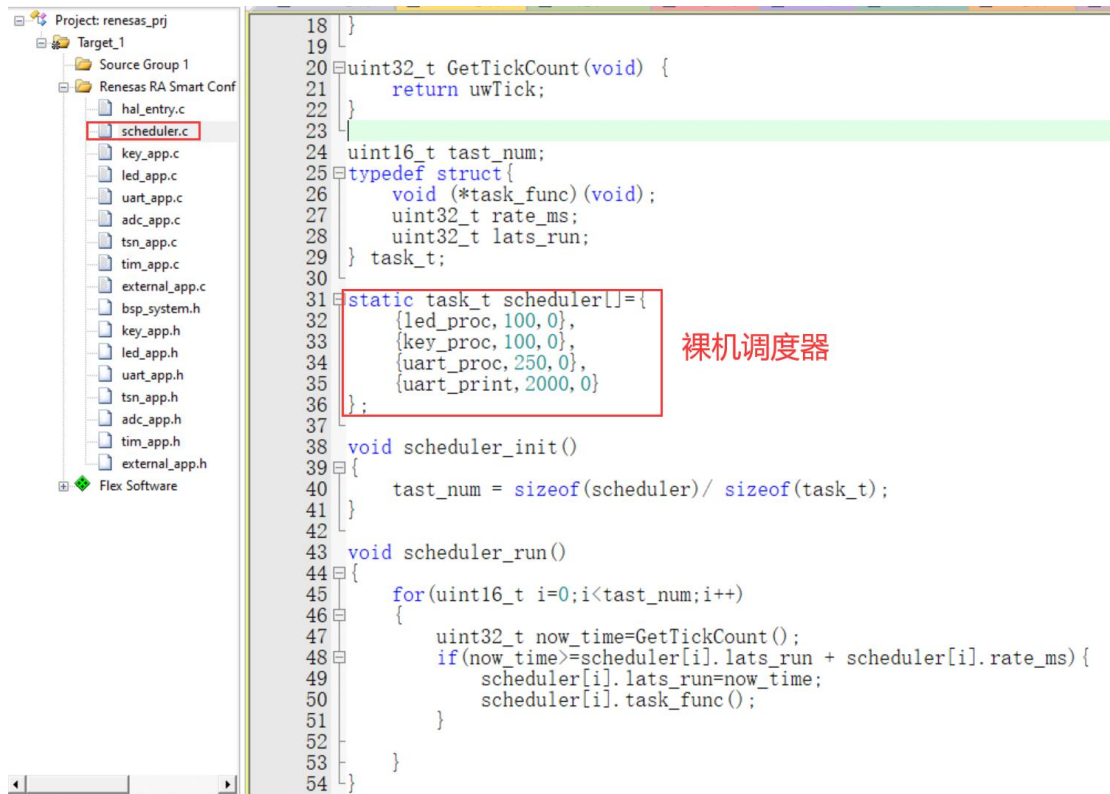


```
1 #include "hal_data.h"
2 #include "stdio.h"
3 #include <stdlib.h>
4 #include "key_app.h"
5 #include "led_app.h"
6 #include "math.h"
7
8 全局变量声明结构体
9 typedef struct{
10     int system_mode;//系统模式
11
12     char uart_buffer[50];//串口数据储存
13
14     float cpu_temper;
15     float adc_data;
16 }system_parameter;
17
18 void scheduler_init();
19 void scheduler_run();
20 void systick_init(uint32_t IT_frequency);//中断初始化
21
22 void UART9_Init(void);
23 void uart_proc();
24 void uart_print();
25
26
27 void ADC_Init(void);
28 float get_adcddata();
29 float temperature_get();
30
31 void GPT_PWM_Init(void);
32 void GPT_Timing_Init(void);
33
34 void P015_IRQ_Init(void);
35
```

项目的配置头文件，包含全局类型定义和函数声明，方便各模块引用。

结构体 system\_parameter 用来声明在整个工程里都需要用到的全局变量，添加新的变量仅需在结构体中继续声明即可。

## 3.2 scheduler



裸机调度器，通过数组创建伪多线程，读取单片机的滴答定时器来实现功能函数定期执行的效果。添加新的线程（即需要定期执行的功能）需要在数组 **scheduler** 中创建新的元素。

格式：{需要执行的函数名，每隔多少时间执行一次（单位 ms），第一次执行的时间（单位 ms）}。

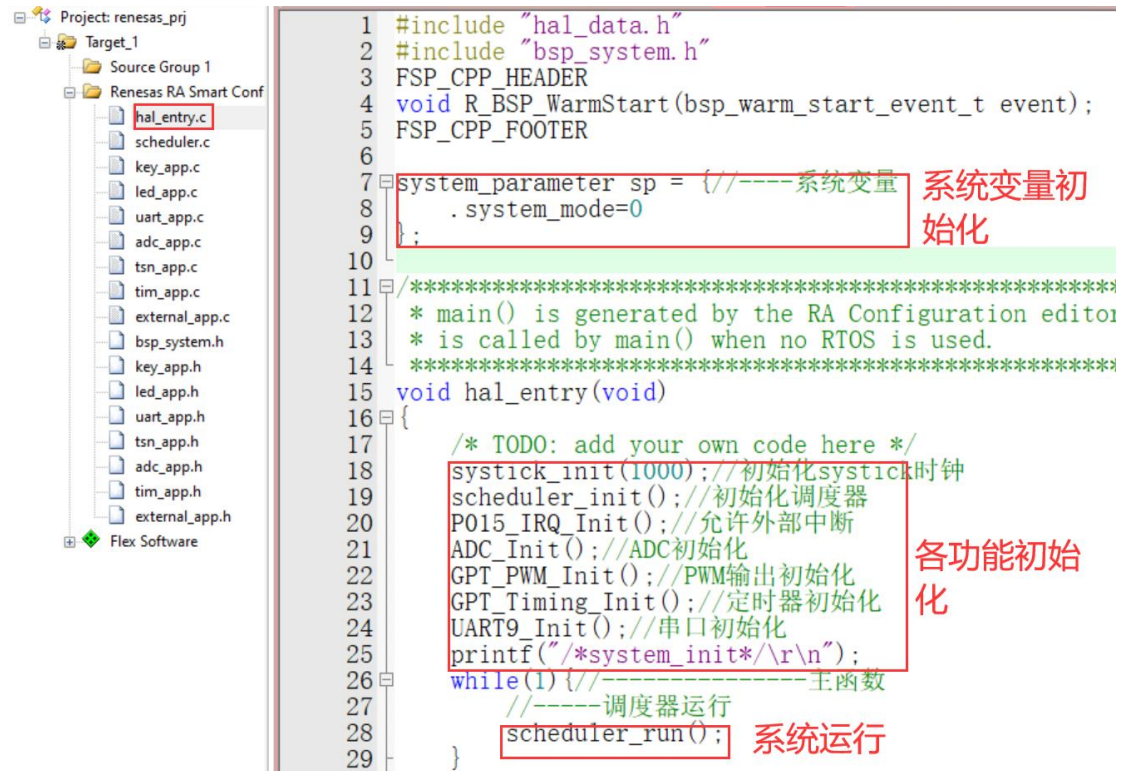
例：{uart\_proc, 250, 0}，意思是 uart\_proc 函数，从 scheduler\_run 函数开始执行的第 0ms 开始，每 250ms 执行一次。

函数功能：

**scheduler\_init**：读取需要运行的线程数。

**scheduler\_run**：用于遍历任务调度器中的任务列表，并根据当前时间判断是否需要执行任务。如果当前时间已超过任务的上次运行时间加上任务的运行间隔，则更新任务的上次运行时间并调用对应的任务函数。

### 3.3 hal\_entry



```
1 #include "hal_data.h"
2 #include "bsp_system.h"
3 FSP_CPP_HEADER
4 void R_BSP_WarmStart(bsp_warm_start_event_t event);
5 FSP_CPP_FOOTER
6
7 system_parameter sp = { //----系统变量 系统变量初
8     .system_mode=0      始化
9 };
10
11 /*
12  * main() is generated by the RA Configuration editor
13  * is called by main() when no RTOS is used.
14  */
15 void hal_entry(void)
16 {
17     /* TODO: add your own code here */
18     systick_init(1000); //初始化systick时钟
19     scheduler_init(); //初始化调度器
20     P015_IRQ_Init(); //允许外部中断
21     ADC_Init(); //ADC初始化
22     GPT_PWM_Init(); //PWM输出初始化
23     GPT_Timing_Init(); //定时器初始化
24     UART9_Init(); //串口初始化
25     printf("/*system_init*/\r\n");
26     while(1) { //-----主函数
27         //-----调度器运行
28         scheduler_run(); 系统运行
29     }
```

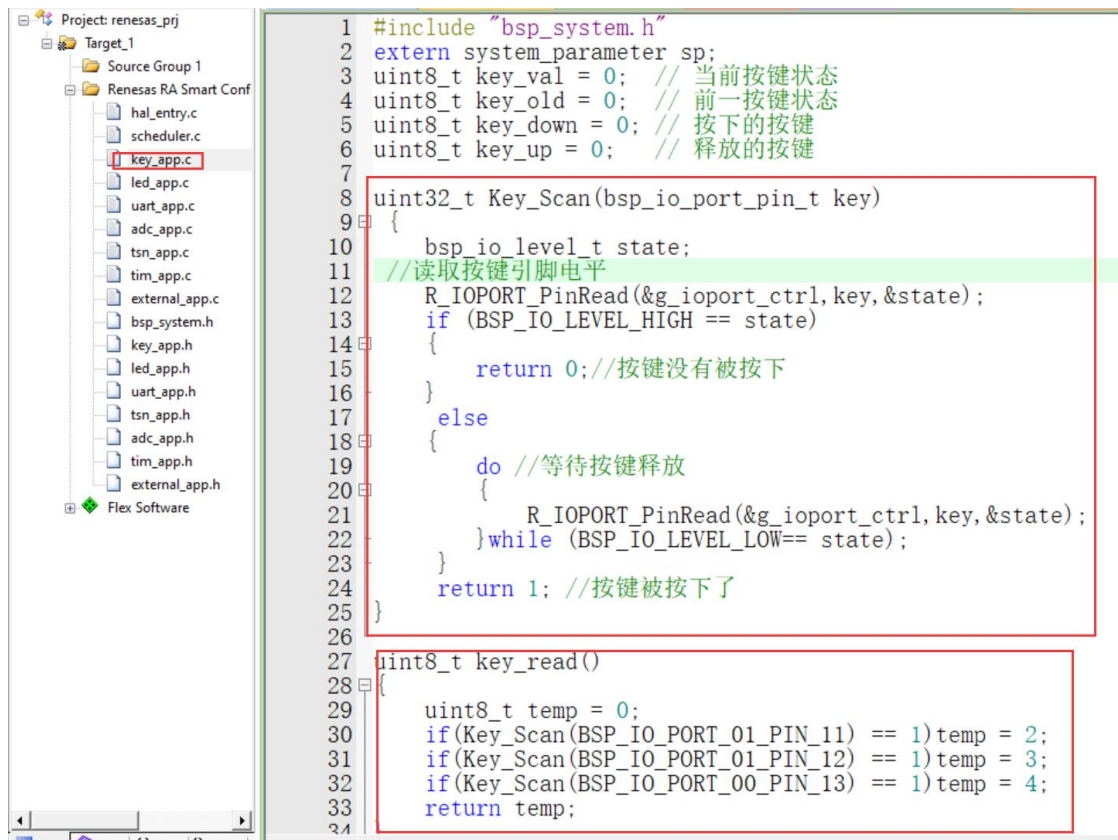
主程序入口，实现系统初始化和主循环调度功能。

初始化系统全局参数结构 sp（定义 system\_parameter sp 并设置初始值）。

初始化完成后，主函数打印系统初始化完成信息，然后进入 while(1) 死循环调用 scheduler\_run()。调度器将以设定的周期反复调用各任务，实现主循环的功能。



### 3.4 key\_app



The screenshot shows an IDE with a project named 'Project: renesas\_prj'. The left sidebar displays the project structure, including 'Target\_1', 'Source Group 1', and 'Renesas RA Smart Conf'. The 'key\_app.c' file is highlighted in the 'Source Group 1' folder. The main editor window displays the source code for 'key\_app.c'. The code includes a header file 'bsp\_system.h' and defines several global variables: 'key\_val', 'key\_old', 'key\_down', and 'key\_up'. It also defines a function 'Key\_Scan' that takes a pin number as input and returns a value. The function 'Key\_Scan' is implemented with a loop that reads the pin state and returns 0 if the key is not pressed, or 1 if it is. The 'key\_read' function is also defined, which calls 'Key\_Scan' for three different pins and returns the result.

```
1 #include "bsp_system.h"
2 extern system_parameter sp;
3 uint8_t key_val = 0; // 当前按键状态
4 uint8_t key_old = 0; // 前一按键状态
5 uint8_t key_down = 0; // 按下的按键
6 uint8_t key_up = 0; // 释放的按键
7
8 uint32_t Key_Scan(bsp_io_port_pin_t key)
9 {
10     bsp_io_level_t state;
11     //读取按键引脚电平
12     R_IOPORT_PinRead(&g_ioport_ctrl, key, &state);
13     if (BSP_IO_LEVEL_HIGH == state)
14     {
15         return 0; //按键没有被按下
16     }
17     else
18     {
19         do //等待按键释放
20         {
21             R_IOPORT_PinRead(&g_ioport_ctrl, key, &state);
22             } while (BSP_IO_LEVEL_LOW == state);
23     }
24     return 1; //按键被按下了
25 }
26
27 uint8_t key_read()
28 {
29     uint8_t temp = 0;
30     if (Key_Scan(BSP_IO_PORT_01_PIN_11) == 1) temp = 2;
31     if (Key_Scan(BSP_IO_PORT_01_PIN_12) == 1) temp = 3;
32     if (Key_Scan(BSP_IO_PORT_00_PIN_13) == 1) temp = 4;
33     return temp;
34 }
```

```

void key_proc()
{
    // 读取当前按键状态
    key_val = key_read();
    // 计算按下的按键（当前按下状态与前一状态异或，并与当前状态相与）
    key_down = key_val & (key_old ^ key_val);
    // 计算释放的按键（当前未按下状态与前一状态异或，并与前一状态相与）
    key_up = ~key_val & (key_old ^ key_val);
    // 更新前一按键状态
    key_old = key_val;

    switch(key_down) {
        case 2:
            /*功能：K2按下时 sp.system_mode加一 最多到2*/
            if(++sp.system_mode==2) sp.system_mode=0;
            break;
        case 3:

            break;
        case 4:

            break;
    }
}

```

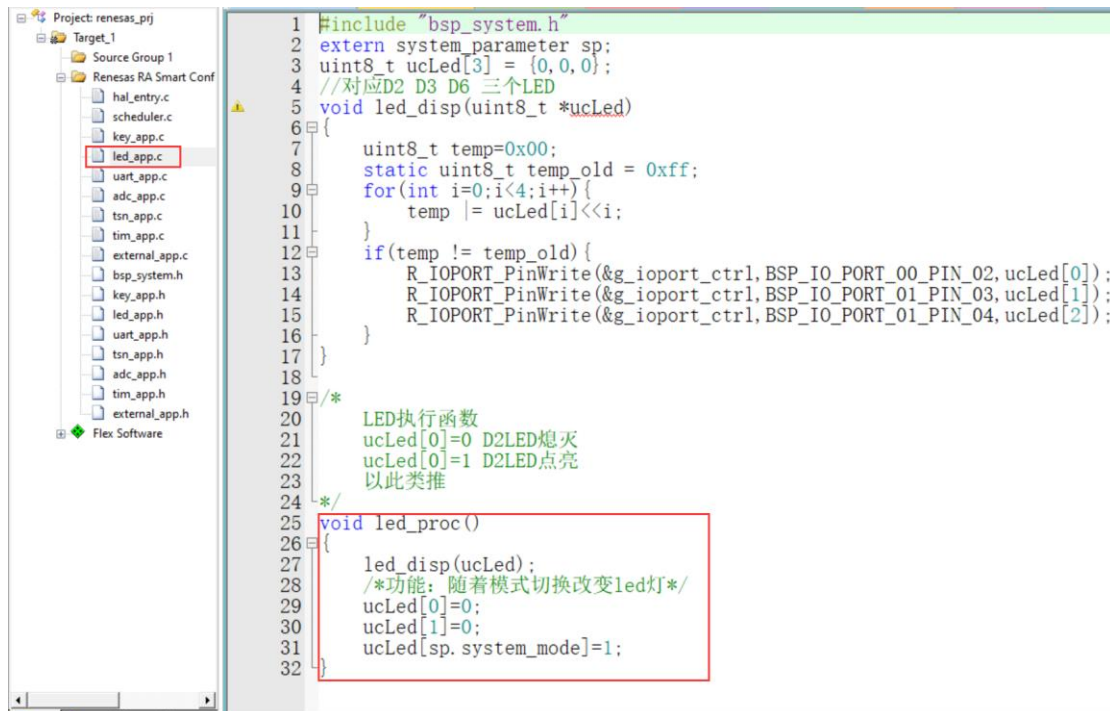
负责按键扫描和模式切换。

硬件有三个按键（K2、K3、K4）接入特定引脚并配置上拉，按下时输出低电平。该模块通过定期扫描按键引脚电平，实现对按键按下、释放和长按的检测，并据此触发系统模式的改变等操作。

**函数功能：**

**key\_proc:** 100ms 由调度器执行一次。在本工程中，按键采集的是**松开沿事件**，通过 K2、K3、K4 的弹起返回 2，3，4 来对按下的按键进行判断并进行对应的处理。使用时通过修改 switch 函数后对应的 case 2，3，4 的内容来修改按键事件。

### 3.5 led\_app



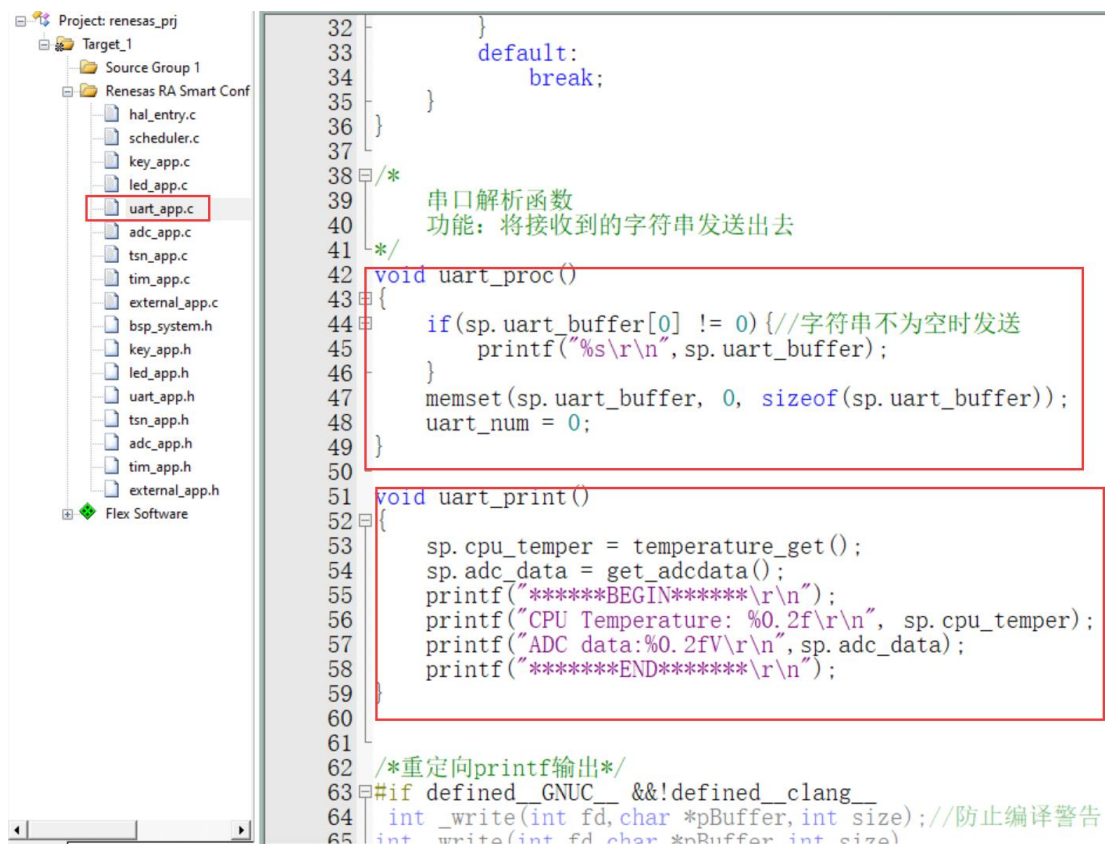
```
1 #include "bsp_system.h"
2 extern system_parameter sp;
3 uint8_t ucLed[3] = {0, 0, 0};
4 //对应D2 D3 D6 三个LED
5 void led_disp(uint8_t *ucLed)
6 {
7     uint8_t temp=0x00;
8     static uint8_t temp_old = 0xff;
9     for(int i=0;i<4;i++){
10         temp |= ucLed[i]<<i;
11     }
12     if(temp != temp_old){
13         R_IOPORT_PinWrite(&_ioport_ctrl, BSP_IO_PORT_00_PIN_02, ucLed[0]);
14         R_IOPORT_PinWrite(&_ioport_ctrl, BSP_IO_PORT_01_PIN_03, ucLed[1]);
15         R_IOPORT_PinWrite(&_ioport_ctrl, BSP_IO_PORT_01_PIN_04, ucLed[2]);
16     }
17 }
18
19 /*
20  LED执行函数
21  ucLed[0]=0 D2LED熄灭
22  ucLed[0]=1 D2LED点亮
23  以此类推
24 */
25 void led_proc()
26 {
27     led_disp(ucLed);
28     /*功能： 随着模式切换改变led灯*/
29     ucLed[0]=0;
30     ucLed[1]=0;
31     ucLed[sp.system_mode]=1;
32 }
```

负责板上 LED 指示灯的控制逻辑

函数功能：

**led\_proc:** 100ms 由调度器执行一次。全局数组 ucLed[3]用于更改三个 LED 的目标状态。从左到右分别为 D2 D3 D6 通过更改 ucLed[3]中的元素来控制 LED 的状态。例如 ucLed[0]=0 设置 D2 为灭，ucLed[0]=1 设置 D2 为亮。

### 3.6 uart\_app



负责串行通信 UART 的初始化和收发处理。串口 USB 接口为板子右上角的 USB 口，波特率为 115200。

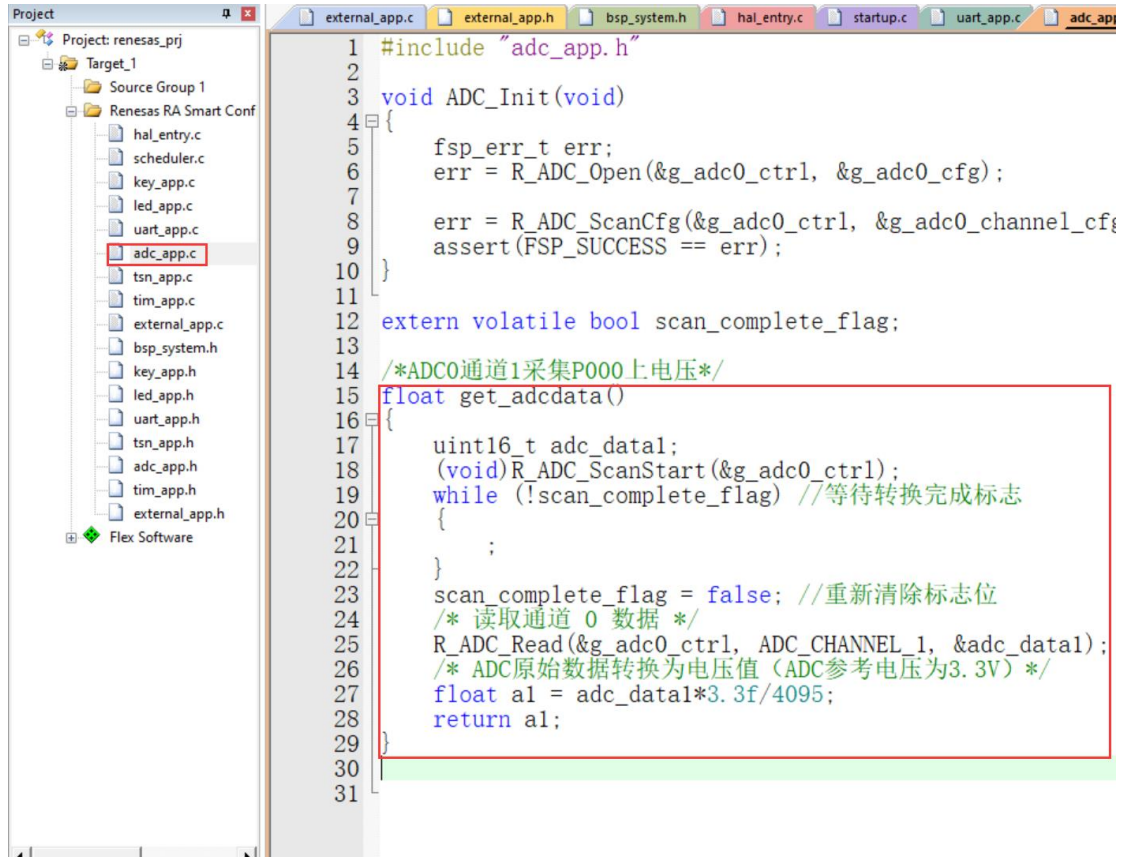
**函数功能：**

**uart\_proc:** 250ms 由调度器执行一次。UART 接收处理函数，被调度器周期性调用，用于处理收到的完整命令或数据。在实现中，它简单地判断 UART 缓冲是（`sp. uart_buffer[0] != 0`），如果有数据则调用 `printf` 将该缓冲内容发送回串口（实现回显或将收到的信息打印出来）。然后清空 `sp. uart_buffer` 并重置计数索引 `uart_num`。这样，每当有新串口数据到达，`uart_proc()` 会在下一个周期将其打印输出，清除缓冲以待下一次接收。

**uart\_print:** 2000ms 由调度器执行一次。用于定时发送系统状态，。其功能是读取当前系统参数中的 CPU 温度和 ADC 电压值，然后通过多次 `printf` 将这些信息格式化输出到串口终端。比如按顺序打印“CPU Temperature: xx.xx”和“ADC data: yy.yyV”等内容。

**printf:** 重定向后的串口发送函数。

## 3.7 adc\_app



负责模数转换初始化和数据采集。

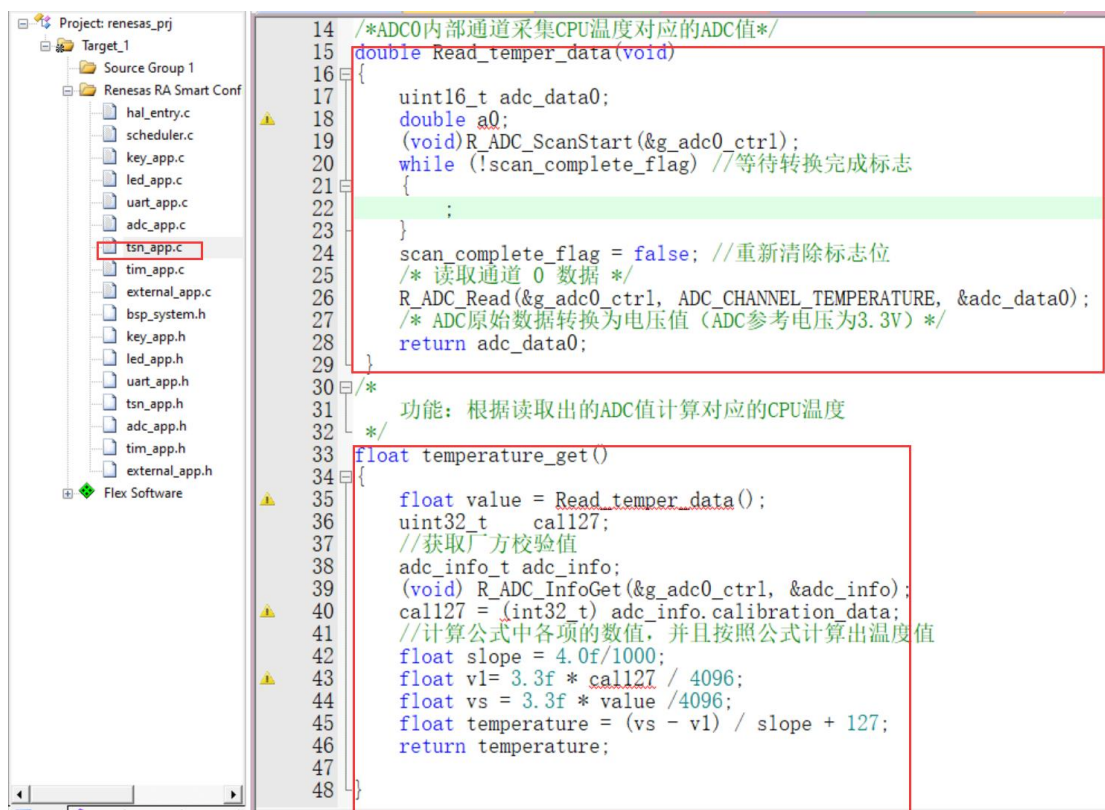
函数功能：

**ADC\_Init:** 用于配置并启动模数转换器 ADC0。

**get\_adcddata:** 通过 R\_ADC\_ScanStart() 启动一次 ADC 扫描转换，然后等待转换完成标志 scan\_complete\_flag 置位。该标志为全局 volatile bool 变量，在 ADC 转换结束的中断回调中被置为 true。当标志变为 true 时，函数清除标志并调用 R\_ADC\_Read() 读取指定通道（如通道 1）的 ADC 结果。代码将返回的原始数字量（0~4095 的 12 位值）转换为 0~3.3V 的实际电压。



## 3.8 tsn\_app

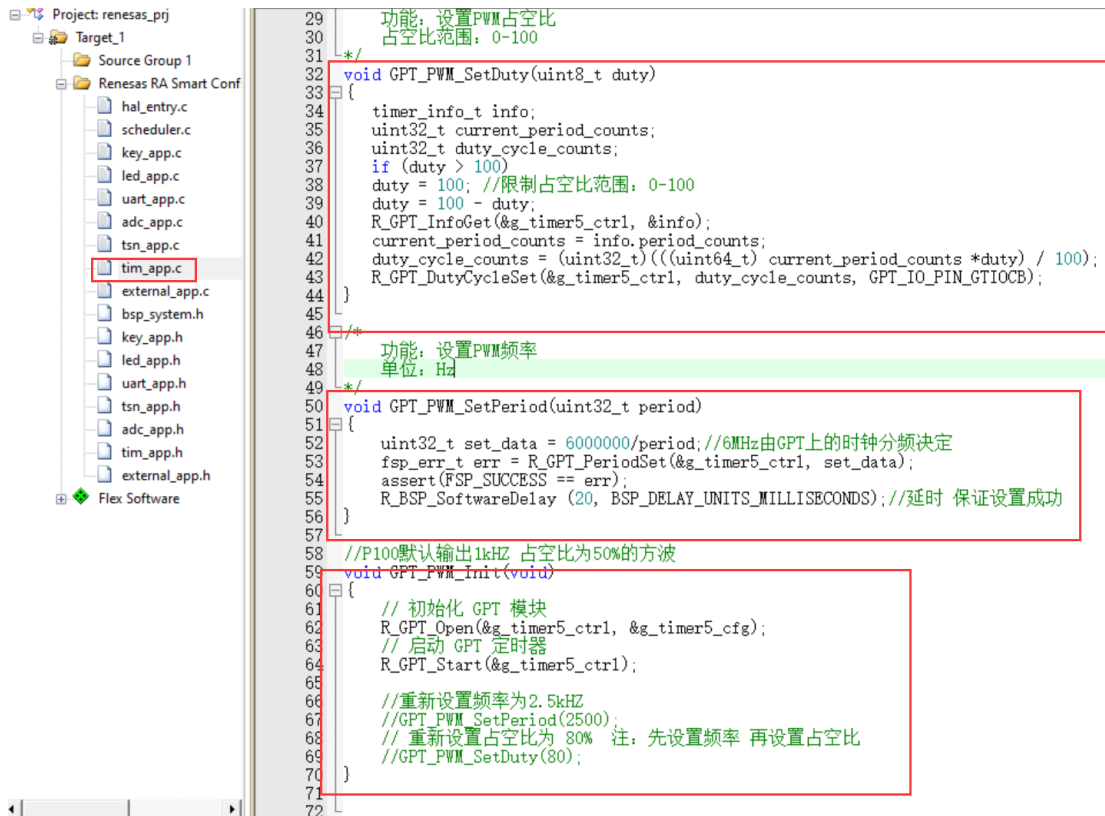


用于读取芯片内部温度传感器的数据，并转换为摄氏温度值。

函数功能：

**temperature\_get：**获取当前芯片温度。内部先调 Read\_temper\_data() 执行一次温度采样：启动 ADC 转换并等待完成标志，然后调用 R\_ADC\_Read() 读取 ADC 的温度通道原始数据（通常 ADC\_CHANNEL\_TEMPERATURE）。获得原始数据后，temperature\_get() 再根据芯片的标定值计算实际温度。具体而言，RA4 系列 MCU 通常在生产时于 127° C 有一个校准基准值。代码通过 R\_ADC\_InfoGet() 获取 ADC 校准数据，即 127°C 时的 ADC 读数 calibration\_data。然后按公式计算温度：先计算出校准温度下的电压  $v1 = 3.3 * cal127 / 4096$ ，再计算当前读数对应电压  $vs = 3.3 * value / 4096$ （4096 代表 ADC 全量程）。已知传感器输出电压随温度变化斜率约为 4.0 mV/°C（即  $slope = 4.0f/1000$ ），于是温度 =  $(vs - v1) / slope + 127$ 。函数最终返回计算出的摄氏温度值。

## 3.9 tim\_app



封装了 GPT 定时器的使用，包括一个定时中断定时器和一个 PWM 输出定时器的配置与控制。

函数功能：

**GPT\_Timing\_Init:** 初始化基本定时器 GPT0，使其产生周期性中断。该函数调用 R\_GPT\_Open() 打开 GPT0 通道（利用全局控制结构 g\_timer0\_ctrl 和配置 g\_timer0\_cfg），然后调用 R\_GPT\_Start() 启动定时器 [GitHub](#)。按照配置，GPT0 被设置为 1 毫秒周期中断。

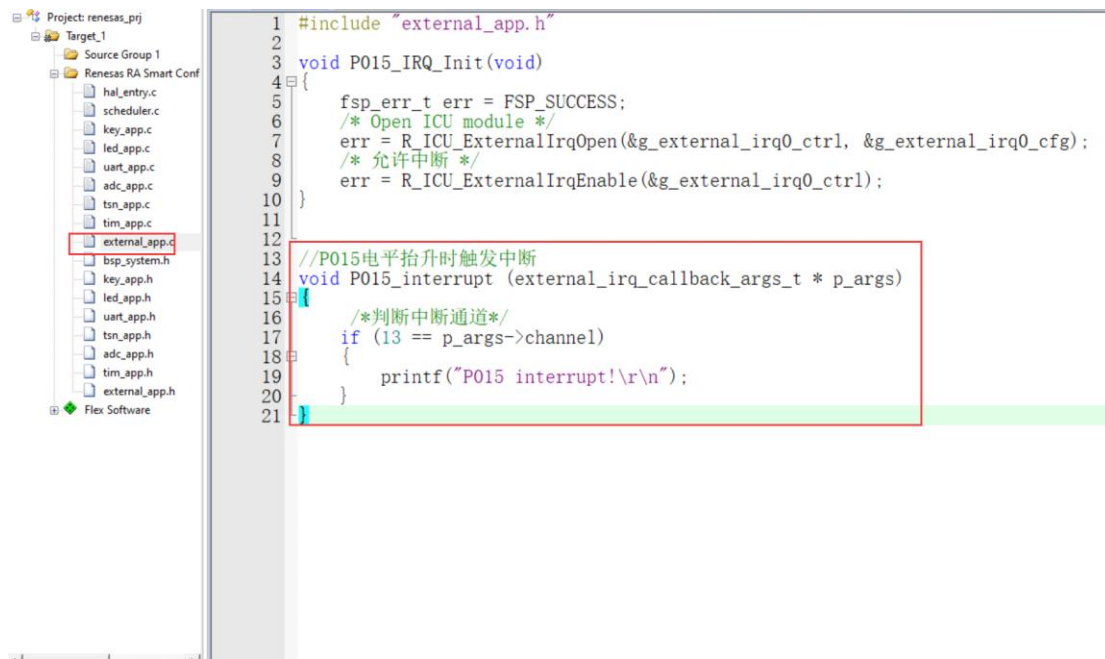
**gpt0\_callback:** GPT0 的中断回调函数。在每次定时器周期结束（溢出）时，该回调执行：代码使用一个静态计数器 timer\_500ms 累加每毫秒触发次数，当计数达到 500（即 500ms）时，将数组 ucLed[2] 对应的 LED 状态取反。

**GPT\_PWM\_Init:** 初始化 GPT5：调用 R\_GPT\_Open() 配置 GPT5 定时器（如设定初始频率 1kHz，占空比 50% 输出到指定引脚，例如 P1.00），然后 R\_GPT\_Start() 启动 PWM 输出。

**GPT\_PWM\_SetDuty:** 用于设置 PWM 占空比（0-100%）。

**GPT\_PWM\_SetPeriod:** 用于调整 PWM 频率。

## 3.10 external\_app



负责配置和处理外部引脚中断。

函数功能：

**P015\_IRQ\_Init:** 用于初始化该外部中断。

**P015\_interrupt:** 当 P015 引脚发生设定的中断事件（上升沿）时，调用此回调，调用 `printf("P015 interrupt!\r\n")` 输出一条消息提示。