Smart Hardware Design

Conception de matériel intelligent

Diseño de hardware inteligente

スマートハードウェア設計

智能硬件设计

第三章 智能硬件的基础外设

Design de hardware inteligente

تصميم الأجهزة الذكية

Slimme hardwareontwerpen

Σχεδίαση έξυπνου υλικού

大连理工大学-朱明

Progettazione di hardware intelligente

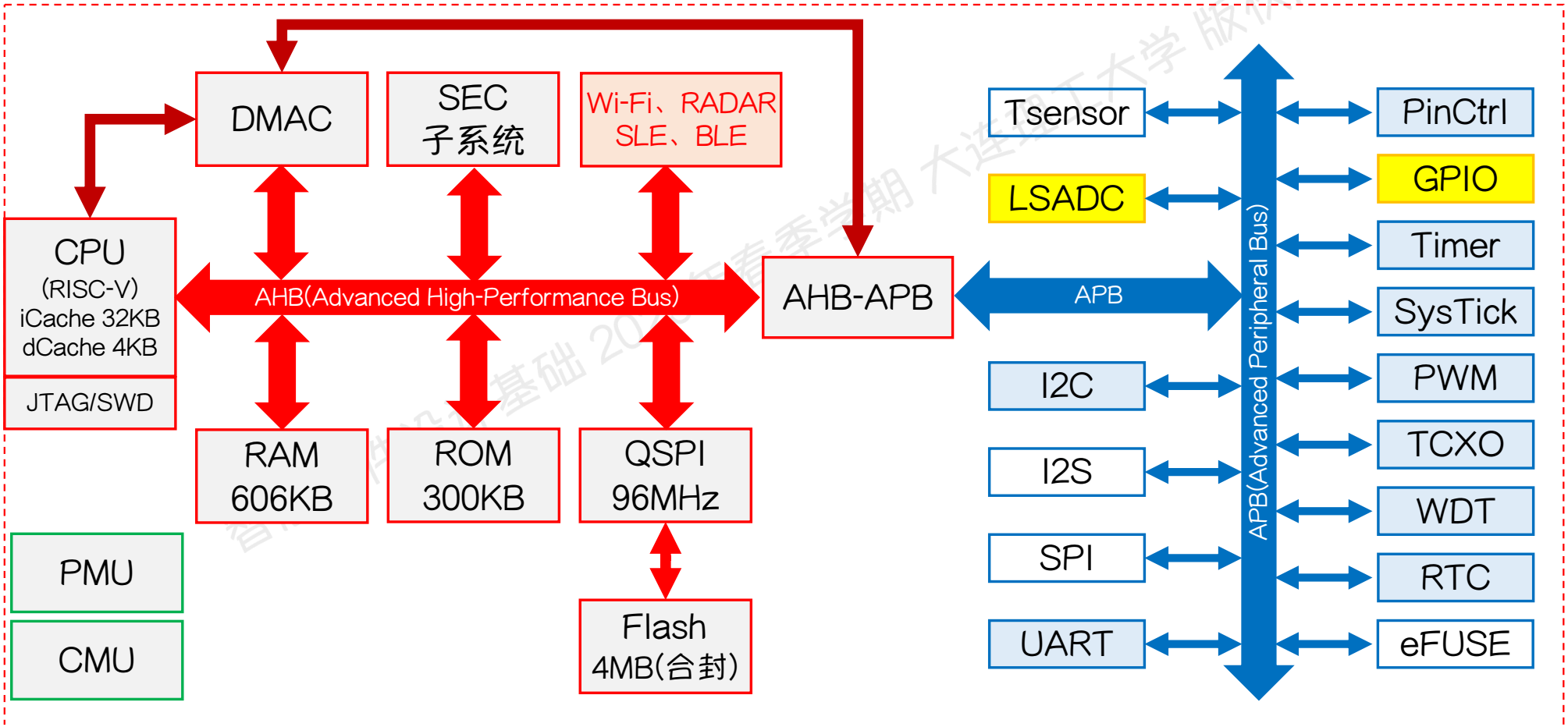
스마트 하드웨어 설계

Smart-Hardware-Design

Проектирование умного оборудования

3.0 思考回顾

- [3.0.0] 智能硬件的SoC内部结构(华为海思WS63)

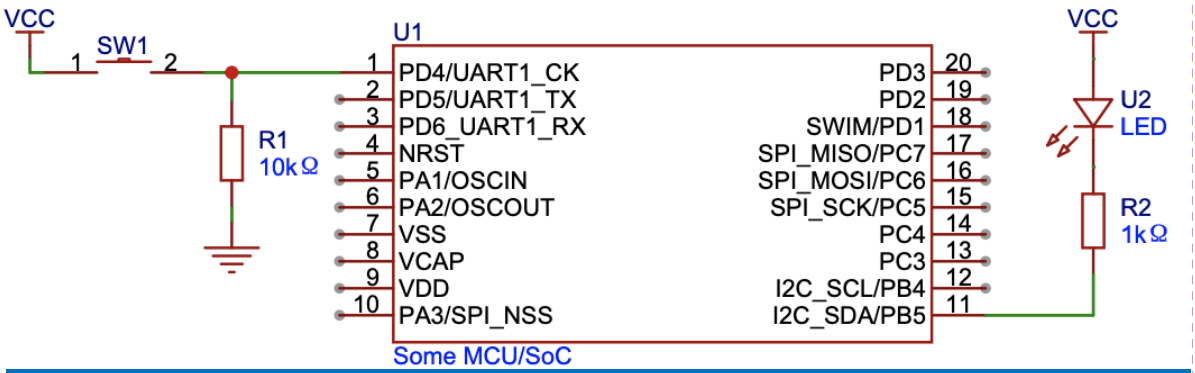


3.0 思考回顾

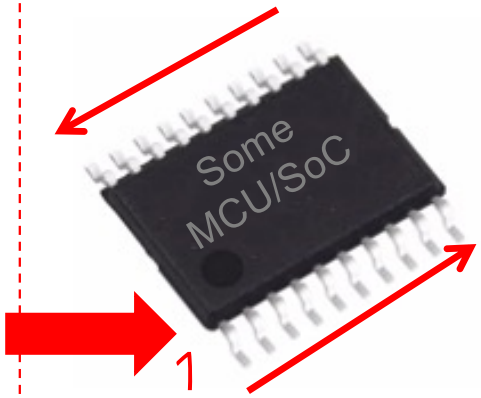
●[3.0.0] 一个简单的SoC的电路

```
#define PD4 (*((volatile unsigned int *)0x40011404))
#define PB5 (*((volatile unsigned int *) 0x40010C08))

int main() {
    while(1) {
        if(PD4 == 1) {      //按键被按下
            PB5 = 0;        //LED点亮
        } else {            //按键没有被按下
            PB5 = 1;        //LED熄灭
        }
    }
}
```



第一层封装：寄存器访问；第二层封装：API访问



按键状态	PD4输入电压	PD4输入逻辑
按键松开	0V	低电平(0)
按键按下	VCC(3.3V)	高电平(1)

PB5输出逻辑	PB5输出电压	LED状态
低电平(0)	0V	点亮
高电平(1)	VCC(3.3V)	熄灭

可以被MCU/SoC检测的按键的状态

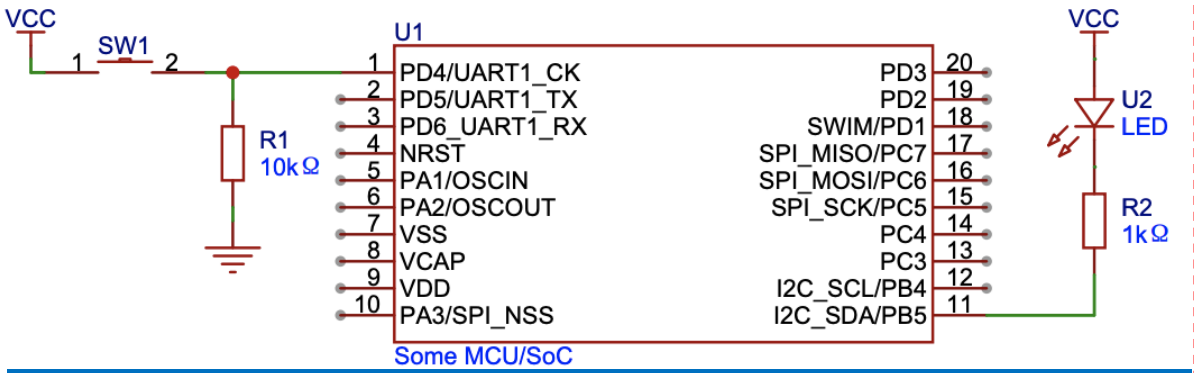
可以被MCU/SoC控制的LED的状态

3.0 思考回顾

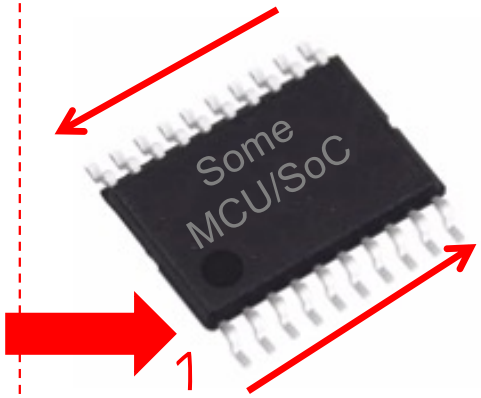
●[3.0.0] 一个简单的SoC的电路

```
#define PD4 (*((volatile unsigned int *)0x40011404))
#define PB5 (*((volatile unsigned int *) 0x40010C08))

int main() {
    while(1) {
        if(PD4 == 1) {      //按键被按下
            PB5 = 0;        //LED点亮
        } else {            //按键没有被按下
            PB5 = 1;        //LED熄灭
        }
    }
}
```



第一层封装：寄存器访问；第二层封装：API访问



按键状态	PD4输入电压	PD4输入逻辑
按键松开	0V	低电平(0)
按键按下	VCC(3.3V)	高电平(1)

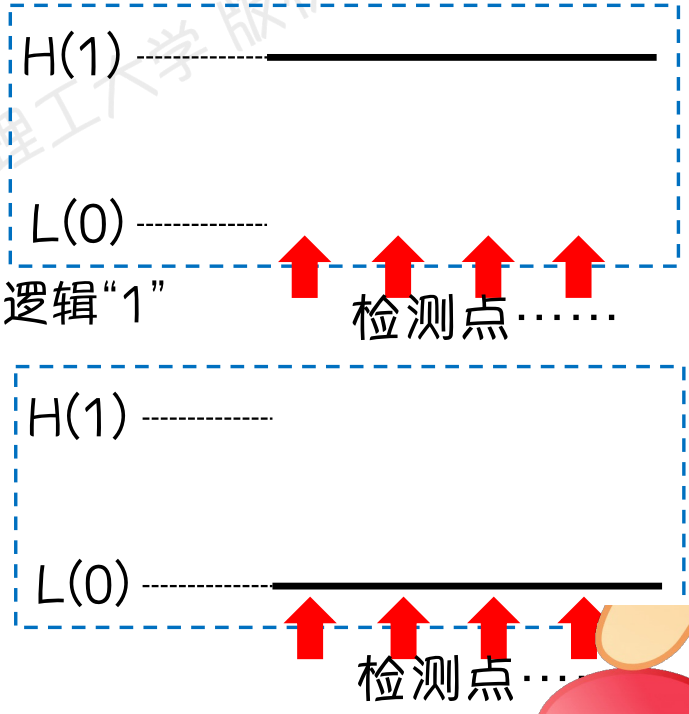
PB5输出逻辑	PB5输出电压	LED状态
低电平(0)	0V	点亮
高电平(1)	VCC(3.3V)	熄灭

可以被MCU/SoC检测的按键的状态
数字电平输入
可以被MCU/SoC控制的LED的状态
数字电平输出

3.1 通用输入输出(模块)

[3.1.1] 通用输入输出模块(GPIO)的基本特性

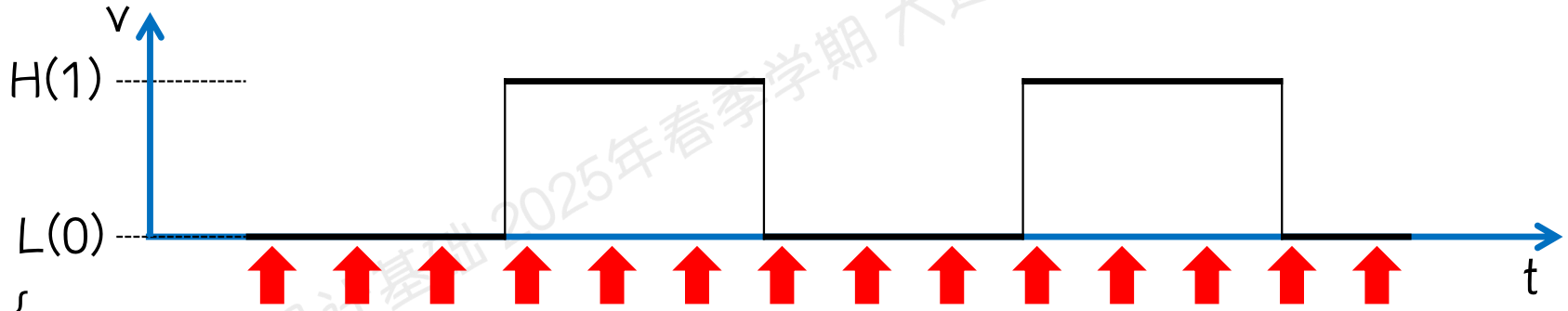
- GPIO(General Purpose Input/Output)
- GPIO具备基本的输入或输出的功能
  - 输入：具备识别外部数字信号的功能
    - 按键按下：电路VCC(3.3V)，高电平信号 -> 逻辑“1”
      - 每次检测，PD4引脚的电压是3.3V
      - 即逻辑电平为高，PD4寄存器的值是1
    - 按键松开：电路0V，低电平信号 -> 逻辑“0”
      - 每次检测，PD4引脚的电压是0V
      - 即逻辑电平为低，PD4寄存器的值是0
  - Q：按下之前是什么状况？松开之后是什么状况？连续起来是什么状



GPIO的输入功能的特征：输入就是获知外部信号的状态，信号是从外至内的

3.1 通用输入输出(模块)

- [3.1.1] 通用输入输出模块(GPIO)的基本特性
  - GPIO具备基本的输入或输出的功能
    - 输入：具备识别外部数字信号的功能




```
while(1) {
    if(PD4==1)
    { …… }
}
```

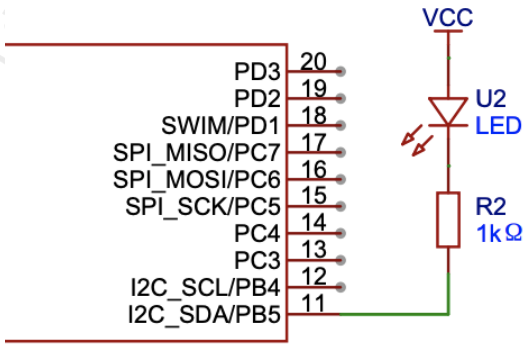
按照时间顺序连续，且可发生变化 > 时序信号 > 时序图 Timing Chart

时序图是进行智能硬件系统的底层硬件/软件开发的最重要基础知识之一

3.1 通用输入输出(模块)

●[3.1.1] 通用输入输出模块(GPIO)的基本特性

- GPIO具备基本的输入或输出的功能
  - 输出：具备向外输出数字信号(电平)的功能
    - PB5逻辑为“1”，输出VCC(3.3V)
      - Q：LED此时是何种状态？为什么是这种状态？
    - PB5逻辑为“0”，输出0V
      - Q：LED此时是何种状态？为什么是这种状态？



LED工作特性

1 单向导电特性：与二极管相同，正向导通、反向截止
2 正向导通电压：普通LED(一般的黄色、绿色和红色)的VF在1.8V左右
3 正常工作电流：普通LED在几mA～几十mA左右
4 亮度调节方法：以较高的速度(1K~10KHz左右)控制LED在工作和不工作状态间切换

LED是电子设备上，最常用的发光器件，用于指示状态、发光照明灯

3.1 通用输入输出(模块)

●[3.1.2] 通用输入输出模块(GPIO)的基本控制

- 寄存器访问方式：读取输入电平状态
  - 课程示例是极少使用的模式
    - 32位长度控制一个功能->浪费
    - 实际上只用到了最后一位

```
#define PD4 (*((volatile unsigned int *)0x40011404))
unsigned int inputValue = 0;
inputValue = PD4;
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- 1=0b00000000000000000000000000000001
- 0=0b00000000000000000000000000000000

前面的31位任何情况下都不会被使用到

为节约资源
其他的位用来
控制其他功能

Q：C语言单独访问其中的某一位，例如读取第7位的值

寄存器是一类重要存储资源，一般每一位都不会被浪费

3.1 通用输入输出(模块)

●[3.1.2] 通用输入输出模块(GPIO)的基本控制

- 寄存器访问方式：读取输入电平状态
  - 按位访问方式

Q：C语言单独访问其中的某一位，例如读取第7位的值

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

```
#define someRegister (*((volatile unsigned int *)0x40011404))
unsigned char inputValue = 0;                                    //保存第七位的值
unsigned int temp;
temp = someRegister;
inputValue = (unsigned char)((temp & (1 << 7)) >> 7);            //填写此处的内容
```

提示：
与操作：&

举例，如何获取第bit位的值
value = ((someRegister & (1 << bit)) >> bit)

位操作是一种低级、高效的操作方式，是硬件编程的最重要操作方法之一
在数据计算、状态设置、存储优化、校验计算和算法设计等方面有重要作用

3.1 通用输入输出(模块)

[3.1.2] 通用输入输出模块(GPIO)的基本控制

- 寄存器访问方式：设置输出电平状态
  - 相同的方式：按位访问方式

```
#define someRegister (*((volatile unsigned int *) 0x40010C08))
someRegister = ____________________________________________;  //第8位设置为1
```

```
#define someRegister (*((volatile unsigned int *) 0x40010C08))
someRegister = ____________________________________________;  //第9位设置为0
```

提示：
与操作：&
或操作：|

置位(设置为1)：someRegister = someRegister | (1<<bit)
或：someRegister |= (1<<bit)
复位(设置为0)：someRegister = someRegister & ~(1<<bit)
或：someRegister &= ~(1<<bit)

位操作是一种低级、高效的操作方式，是硬件编程的最重要操作方法之一
在数据计算、状态设置、存储优化、校验计算和算法设计等方面有重要作用

[3.1.2] 通用输入输出模块(GPIO)的基本控制

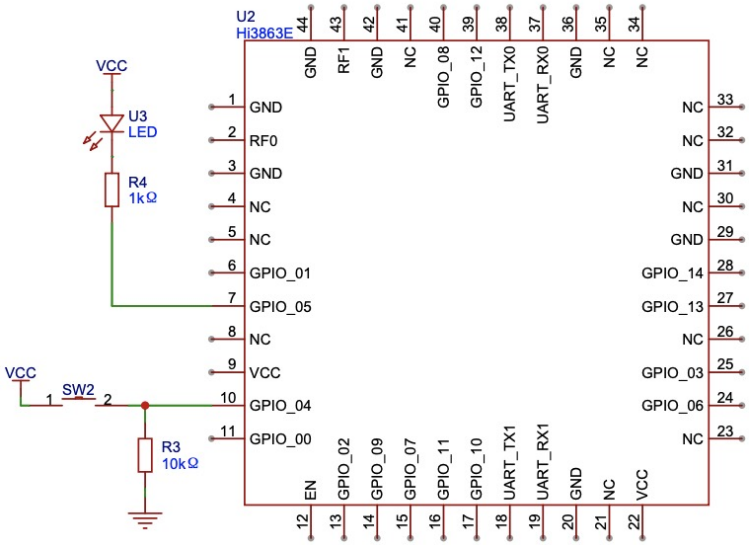
- API访问方式
  - 读取输入电平状态

```
#define   GPIO_04      4
#define   SW2_GPIO     GPIO_04

typedef enum gpio_level {
    GPIO_LEVEL_LOW,
    GPIO_LEVEL_HIGH
} gpio_level_t;
gpio_level_t inputValue;

inputValue = uapi_gpio_get_val(SW2_GPIO);
if(inputValue == GPIO_LEVEL_LOW) { …… }
```

用户完成



上述源码是基于LiteOS系统的WS63 SoC的读取GPIO电平的相关源码

3.1 通用输入输出(模块)

●[3.1.2] 通用输入输出模块(GPIO)的基本控制

● API访问方式

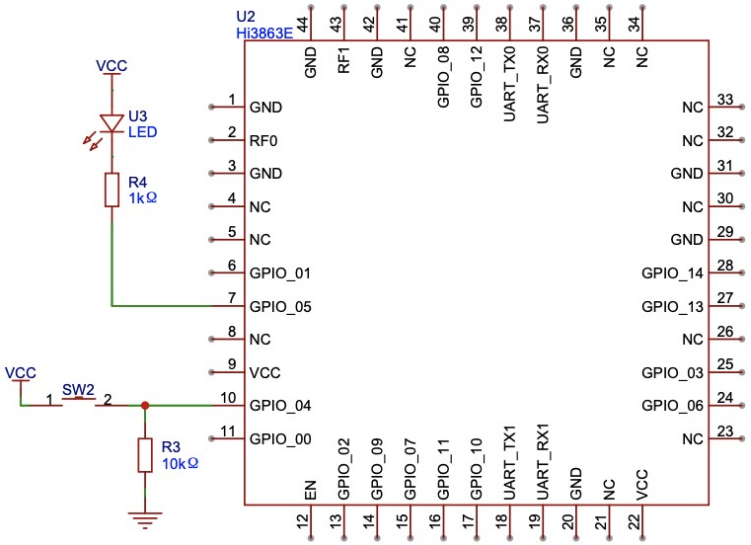
● 设置输出电平状态

```
#define    GPIO_05      5
#define    LED_GPIO     GPIO_05

typedef enum gpio_level {
    GPIO_LEVEL_LOW,
    GPIO_LEVEL_HIGH
} gpio_level_t;
gpio_level_t inputValue;

uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);
uapi_gpio_toggle(LED_GPIO);
```

用户完成



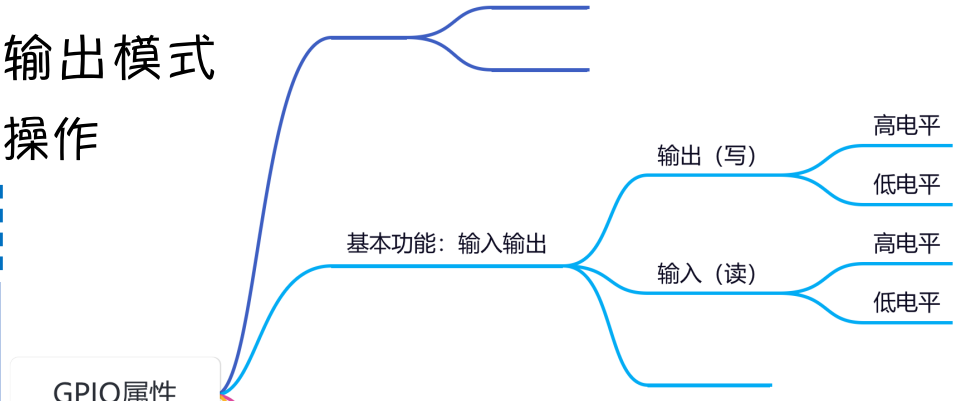
上述源码是基于LiteOS系统的WS63 SoC的设置GPIO电平的相关源码

- [3.1.2] 通用输入输出模块(GPIO)的基本控制
  - GPIO的输入/输出的切换是需要设置的
    - 先设置GPIO工作在输入模式或输出模式
      - 再进行相应的输入或者输出的操作

一般将其命名为GPIO的方向

```
errcode_t uapi_gpio_set_dir(
                    pin_t pin,
                    gpio_direction_t dir);
```

```
typedef enum gpio_direction {
    GPIO_DIRECTION_INPUT,
    GPIO_DIRECTION_OUTPUT
} gpio_direction_t;
```



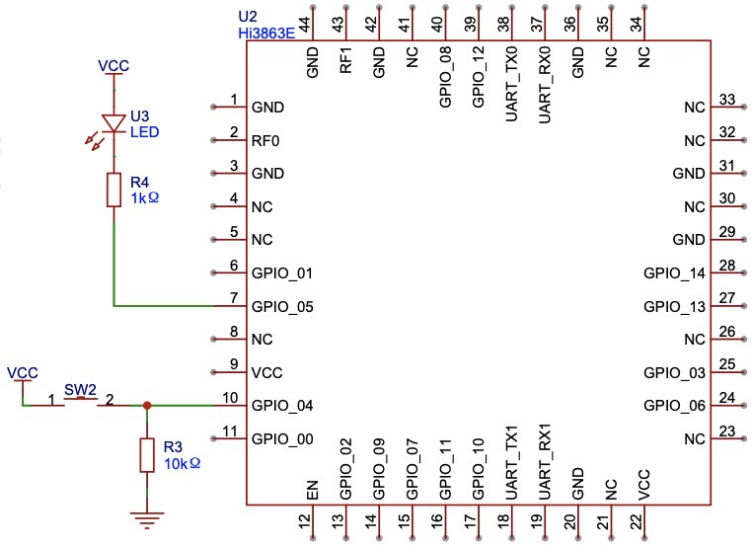

```
uapi_gpio_set_dir(SW2_GPIO, GPIO_DIRECTION_INPUT);
uapi_gpio_set_dir(LED_GPIO, GPIO_DIRECTION_ OUTPUT);
```

Q：用API完成程序

3.1 通用输入输出(模块)

●[3.1.2] 通用输入输出模块(GPIO)的基本控制

```
#define   SW2_GPIO   GPIO_04
#define   LED_GPIO   GPIO_05
static int *keyled_task(const char *arg) {
    unused(arg);
    gpio_level_t inputValue;
    uapi_pin_set_mode(SW2_GPIO, PIN_MODE_2);
    uapi_pin_set_mode(LED_GPIO, PIN_MODE_4);
    uapi_gpio_set_dir(SW2_GPIO, GPIO_DIRECTION_INPUT);
    uapi_gpio_set_dir(LED_GPIO, GPIO_DIRECTION_ OUTPUT);
    while (1) {
        inputValue = uapi_gpio_get_val(SW2_GPIO);
        if(inputValue == GPIO_LEVEL_LOW) {
            uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_HIGH);
        } else {
            uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);
        }
    }
    return 0;
}
```



引脚复用

MCU或SoC的一个物理引脚在使用时，可以设置成多种外设的功能，该功能可以节约引脚资源，减小芯片体积

3.1 通用输入输出(模块)

- [3.1.2] 通用输入输出模块(GPIO)的基本控制
  - GPIO的功能切换是需要设置的
    - 需要设置GPIO引脚的功能

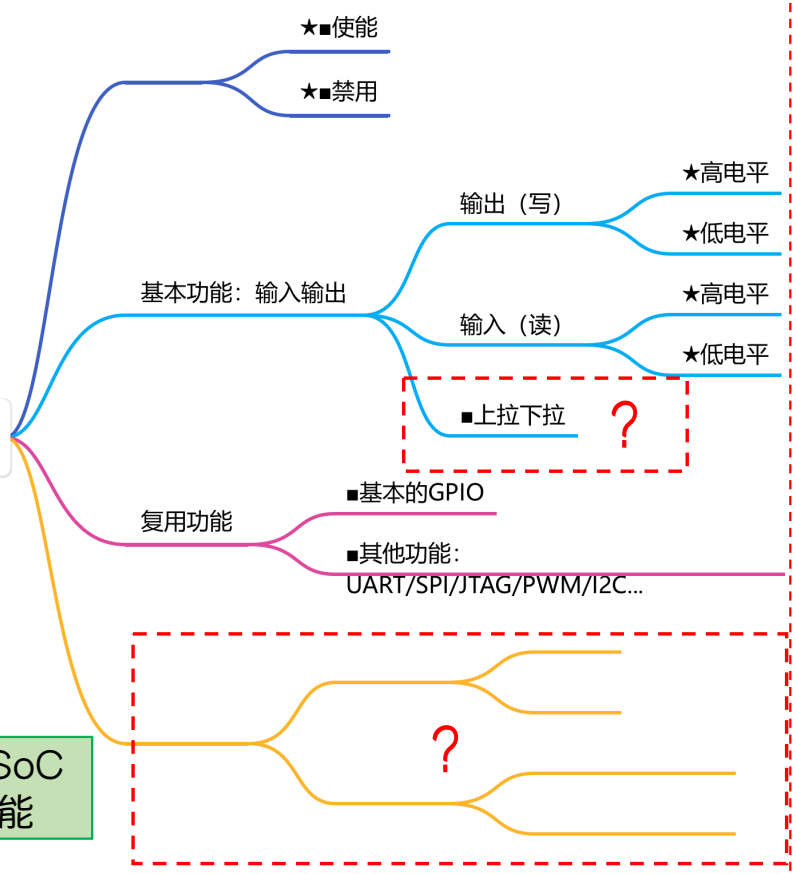
IO/MODE	0	1	2	3	4	5	6
GPIO_00	GPIO_00	PWM0	DIAG[0]	SPI1_CSN	JTAG_TDI		
GPIO_01	GPIO_01	PWM1	DIAG[1]	SPI1_IO0	JTAG_MODE	BT_SAMPLE	
GPIO_02	GPIO_02	PWM2	DIAG[2]	SPI1_IO3	WIFI_TSF_SYNC	WL_GLP_SYNC_PULSE	BGLE_GL
GPIO_03	GPIO_03	PWM3	PMU_32K_TEST	SPI1_IO1	HW_ID[0]	DIAG[3]	
GPIO_04	SSI_CLK	PWM4	GPIO_04	SPI1_IO1	JTAG_ENABLE	DFT_JTAG_TMS	
GPIO_05	SSI_DATA	PWM5	UART2_CTS	SPI1_IO2	GPIO_05	SPI0_IN	DFT_JTA
GPIO_06	GPIO_06	PWM6	UART2_RTS	SPI1_SCK	EFCLK_FREQ_STATUS	DIAG[4]	SPI00_O
GPIO_07	GPIO_07	PWM7	UART2_RXD	SPI0_SCK	I2S_MCLK	DIAG[5]	
GPIO_08	GPIO_08	PWM0	UART2_TXD	SPI0_CS1_N	DIAG[6]		
GPIO_09	GPIO_09	PWM1	RADAR_ANT0_SW	SPI0_OUT	I2S_DO	HW_ID[1]	DIAG[7]
GPIO_10	GPIO_10	PWM2	ANT0_SW	SPI0_CS0_N	I2S_SCLK	DIAG[0]	
GPIO_11	GPIO_11	PWM3	RADAR_ANT1_SW	SPI0_IN	I2S_LRCLK	DIAG[1]	HW_ID[2
GPIO_12	GPIO_12	PWM4	ANT1_SW		I2S_DI		HW_ID[3
GPIO_13	GPIO_13	UART1_CTS	RADAR_ANT0_SW	DFT_JTAG_TD0	JTAG_TMS		
GPIO_14	GPIO_14	UART1_RTS	RADAR_ANT1_SW	DFT_JTAG_TRSTN	JTAG_TCK		
UART1_TXD	GPIO_15	UART1_TXD	I2C1_SDA				
UART1_RXD	GPIO_16	UART1_RXD	I2C1_SCL				
UART0_TXD	GPIO_17	UART0_TXD	I2C0_SDA				
UART0_RXD	GPIO_18	UART0_RXD	I2C0_SCL				

芯片文档中标明复用功能和默认功能，上述表格是WS63 SoC的引脚功能复用表(部分)

3.1 通用输入输出(模块)

[3.1.2] 通用输入输出模块(GPIO)的基本控制

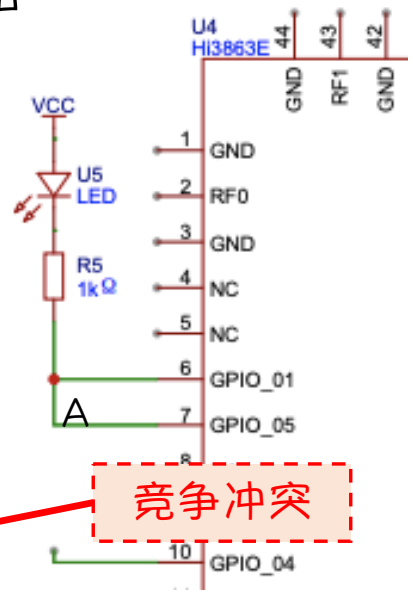
- GPIO的输入输出是需要设置的
  - errcode_t uapi_gpio_set_dir
    (pin_t pin, gpio_direction_t dir)
- GPIO的功能切换(复用)是需要设置的
  - errcode_t uapi_pin_set_mode
    (pin_t pin, pin_mode_t mode)
- GPIO的是否工作是可以受控的
  - void uapi_gpio_init(void)
  - void uapi_gpio_deinit(void)



大多数MCU/SoC
都有类似功能

- [3.1.3] 通用输入输出模块(GPIO)的电气特性
  - GPIO的上拉/下拉是可以受控的
    - 信息系统的目标(经典的信息的含义)：消除不确定性
    - 硬件系统的设计原则：避免电路出现不确定性
      - 不确定性电路：无法明确判定当前状态或输出的数字电路

GPIO_01		GPIO_05		LED状态	A点电平状态
输入		输入		灭	高
输入		输出	0	亮	低
			1	灭	高
输出	0	输入		亮	低
	1			灭	高
输出	0	输出	0	亮	低
	1		1	灭	高
	0		1	未知	未知，短路?
	1		0	未知	



3.1 通用输入输出(模块)

[3.1.3] 通用输入输出模块(GPIO)的电气特性

- GPIO的上拉/下拉是可以受控的
  - 内部上拉/下拉电路的控制：以WS63为例，四种可用模式

```
typedef enum {
    PIN_PULL_TYPE_DISABLE       = 0,        //悬空，仅用于输入确定电平
    PIN_PULL_TYPE_DOWN          = 1,        //下拉
    PIN_PULL_TYPE_STRONG_UP     = 2,        //强上拉
    PIN_PULL_TYPE_UP            = 3,        //上拉
    PIN_PULL_MAX                = 4         //无效
} pin_pull_t;
```

  - errcode_t uapi_pin_set_pull(pin_t pin, pin_pull_t pull_type)
  - pin_pull_t uapi_pin_get_pull(pin_t pin)

上拉/下拉电路仅可用于防止输入悬空，不具有带载能力(一般只能流过mA级别以下的电流)

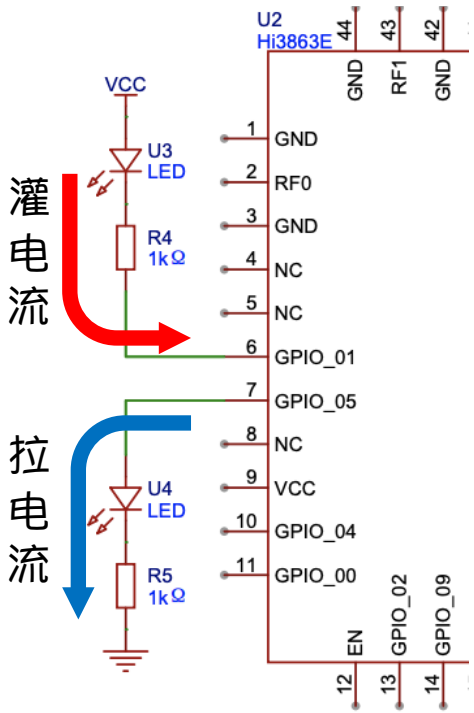
3.1 通用输入输出(模块)

●[3.1.3] 通用输入输出模块(GPIO)的电气特性

- GPIO的电流驱动能力十分有限

输出电压不代表能够输出功率

  - 绝大多数MCU/SoC的GPIO的输出模式下的电流在30mA以下
  - 两种电流流向的最大电流能力不同
  - 灌电流：电流由外部电路流入到芯片内部
    - GPIO的灌电流驱动能力可达20～30mA
    - 直接驱动小高亮LED、小继电器、中功率三极管
  - 拉电流：电流由芯片内部流出到外部电路
    - GPIO的拉电流驱动能力偏小，一般mA级别
    - 驱动普通小LED、小功率三极管



尽管GPIO能直接驱动器件，仍建议使用小功率三极管间接驱动器件

3.1 通用输入输出(模块)

- [3.1.4] WS63的GPIO特性汇总
  - GPIO数量：19个，14个默认为GPIO，4个默认为其他功能
    - 全部支持寄存器访问模式和API访问模式(LiteOS/OpenHarmony支持)
    - 全部可以配置为输入或者输出模式，读取或设置引脚电平状态
    - 全部支持配置为输入模式下的悬空、下拉、强上拉或上拉模式
    - 全部支持使能或禁用功能
    - 全部支持其所在引脚对应复用功能
    - 低电平(L)电压为0V(GND电压)，高电平(H)电压为3.3V(VCC电压)
    - GPIO的输出的拉电流为mA级别，输出的灌电流在10mA级别
    - API模式下GPIO翻转(uapi_gpio_toggle)的执行周期约为1us(非阻塞模式*)

上拉/下拉电路仅可用于防止输入悬空，不具有带载能力(一般只能流过mA级别以下的电流)

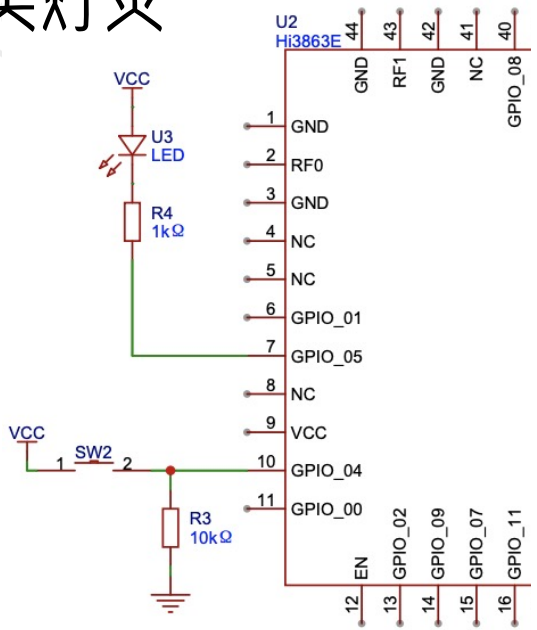
3.2 中断系统

●[3.2.1] 中断系统的应用意义

● 用API方式完成代码，按下开关灯亮，松开开关灯灭

轮询
Polling

```
static int *keyled_task(const char *arg) {
    unused(arg);
    gpio_level_t inputValue;
    uapi_pin_set_mode(SW2_GPIO, PIN_MODE_2);                    //复用：GPIO
    uapi_pin_set_mode(LED_GPIO, PIN_MODE_4);                    //复用：GPIO
    uapi_gpio_set_dir(SW2_GPIO, GPIO_DIRECTION_INPUT);          //方向：输入
    uapi_gpio_set_dir(LED_GPIO, GPIO_DIRECTION_ OUTPUT);        //方向：输出
    while (1) {
        someCalculateFunction();                    //某计算耗时较长时间，如5秒
        inputValue = uapi_gpio_get_val(SW2_GPIO);                //读输入
        if(inputValue == GPIO_LEVEL_LOW) {                       //按键松开
            uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_HIGH);        //LED灭
        } else {                                                 //按键按下
            uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);         //LED亮
        }
    }
    return 0;
}
```



频繁查询寄存器会占用大量时间，耗时长的功能又会阻碍查询工作、出现响应故障

3.2 中断系统

- [3.2.1] 中断(Interrupt)系统的应用目标
  - 提高系统响应效率
    - 允许处理器在需要时才响应，无须轮训设备状态
  - 实现实时性
    - 保证了对高优先级任务的快速响应
  - 降低智能硬件系统设计的复杂性
    - 简化了系统对异步事件的处理过程，程序实现简单
  - 提升系统资源利用率、降低系统功耗
    - 无须轮询和等待，系统可以随时进入低功耗状态
  - 支持系统多任务运行
    - 提供了多任务切换的基础，提供了任务调度的基础

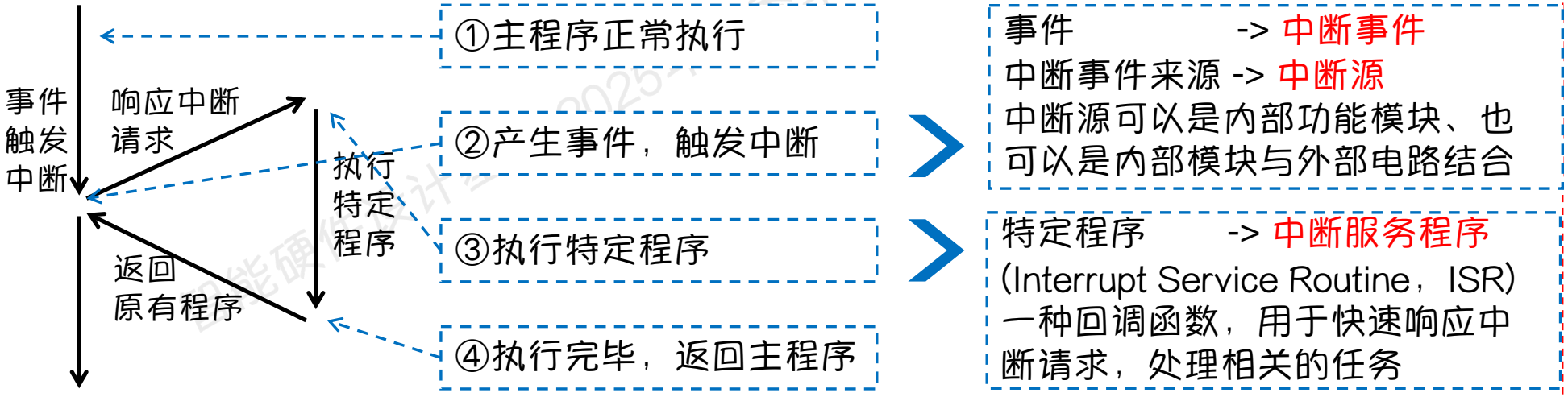
中断是现代信息系统必不可少的工作机制之一

芯片设计：
中断硬件系统等
操作系统：
硬件响应机制等
应用开发：
中断应用开发等
Java：
有中断机制可开发
Python：
有中断机制可开发
……

3.2 中断系统

●[3.2.2] 中断的定义和应用

● MCU/SoC的中断：处理器执行程序的过程中，外部或内部的某些事件触发处理器暂时中断当前执行的任务，转去执行与该事件相关的特定程序，处理完事件后再返回继续原任务的过程



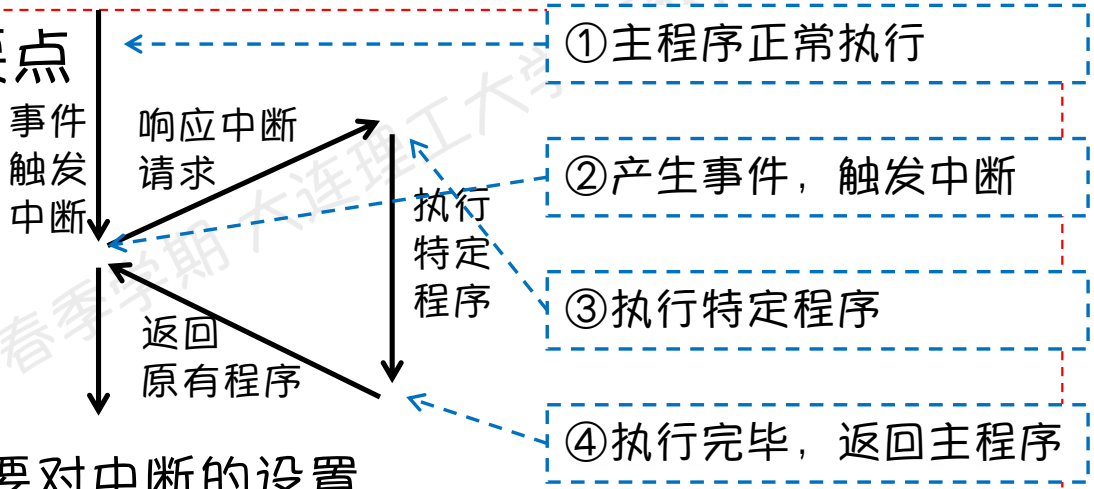
中断的具体实现原理比较复杂，在其他课程中会有详细解释

3.2 中断系统

[3.2.2] 中断的定义和应用

- 在程序中使用单一中断的要点
  - 实现中断服务程序(函数)
  - 实现中断触发的软硬件条件
  - 注册、设置和启用对应中断
  - 触发中断、响应中断
  - 执行中断服务程序，执行必要对中断的设置

系统自动实现



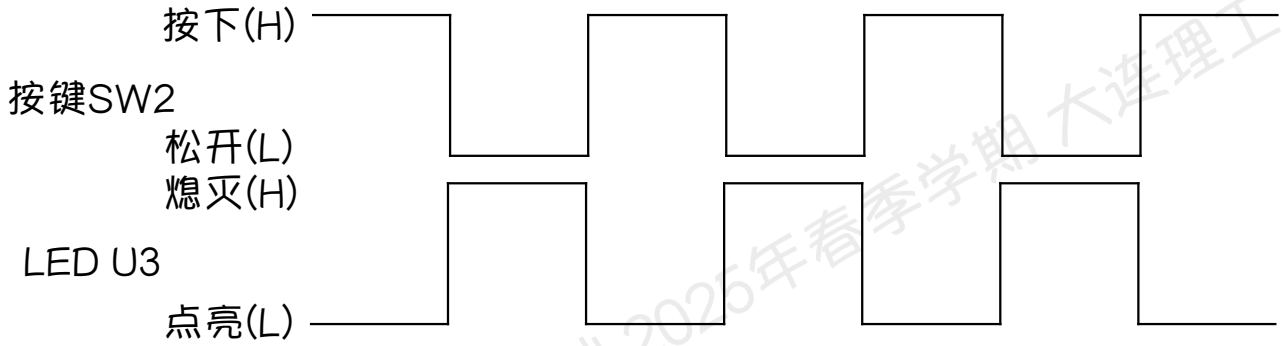
- 在程序中使用中断应避免的问题
  - 避免在中断服务程序(ISR)执行复杂或消耗时间长的工作
  - 避免在中断服务程序(ISR)对资源进行操作，以防与主程序产生冲突

不同的MCU/SoC，以及不同的驱动或系统，使用中断的要点可能会有所差别，基本差异不大

3.2 中断系统

●[3.2.2] 中断的定义和应用

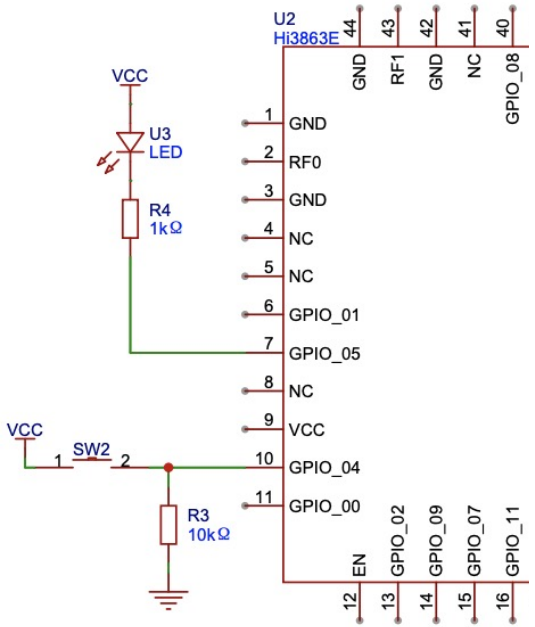
● 单一中断：将LED控制示例改写为中断模式



电平状态

状态变化

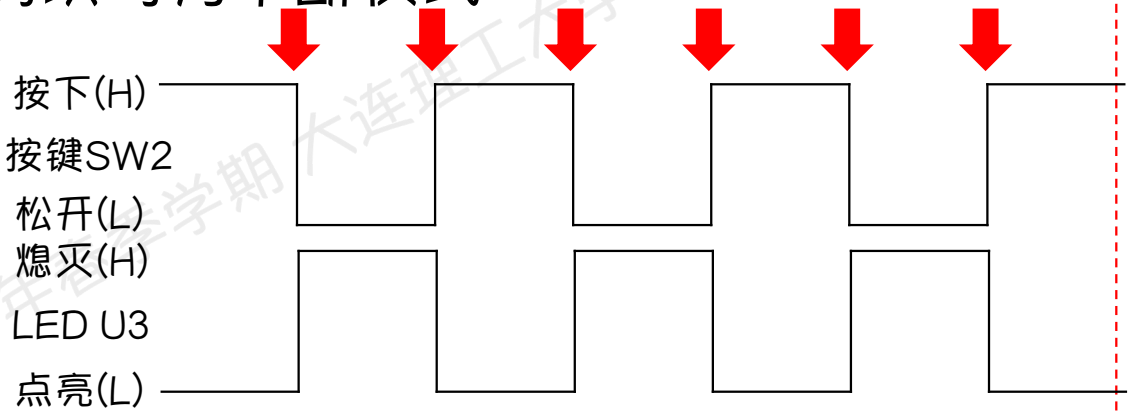
```
inputValue = uapi_gpio_get_val(SW2_GPIO);                    //读输入
if(inputValue == GPIO_LEVEL_LOW) {                           //按键松开
    uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_HIGH);            //LED灭
} else {                                                     //按键按下
    uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);             //LED亮
}
```



考虑一下，是否可以根据电平状态发生变化的事件，去改变LED亮灭

中断的典型优势，将系统资源从频繁查询状态的工作中解脱出来

- [3.2.2] 中断的定义和应用
  - 单一中断：将LED控制示例改写为中断模式
    - 电平变化产生的事件
      - 高电平 -> 低电平
        - 下降沿Falling-edge
      - 低电平 -> 高电平
        - 上升沿Rising-edge



    - 常见MCU/SoC GPIO的中断类型
      - 上升沿触发中断事件：GPIO_INTERRUPT_RISING_EDGE
      - 下降沿触发中断事件：GPIO_INTERRUPT_FALLING_EDGE
      - 上升沿和下降沿都触发中断事件：GPIO_INTERRUPT_DEDGE

中断感知的内容是指定外设的、符合指定类型要求的事件(发生的变化)

3.2 中断系统

●[3.2.2] 中断的定义和应用

● 单一中断：将LED控制示例改写为中断模式(部分源码)

```
static void gpio_callback_func(pin_t pin, uintptr_t param) {
    //……
    inputValue = uapi_gpio_get_val(SW2_GPIO);                    //进入中断后，还需要判断上升沿或下降沿
    if(inputValue == GPIO_LEVEL_LOW) {                           //低电平，按键松开了
        uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_HIGH);            //LED灭
    } else {                                                     //高电平，按键按下了
        uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);             //LED亮
    }
}
static void *button_task(const char *arg) {
    //……                                                        //初始化GPIO、按键和LED初始态等，略
    errcode_t ret = uapi_gpio_register_isr_func(SW2_GPIO, GPIO_INTERRUPT_DEDGE, gpio_callback_func);
    if (ret != 0) {    uapi_gpio_unregister_isr_func(SW2_GPIO);    }
    while (1) {
        uapi_watchdog_kick();                                    //看门狗，后面会讲
        osal_msleep(2000);                                       //延时，模拟其他任务占用CPU
```

中断的引入，大幅度降低了按键查询对系统资源的占用，降低耦合，提高内聚，简化系统开发

3.2 中断系统

●[3.2.2] 中断的定义和应用

● 单一中断：将LED控制示例改写为中断模式(部分源码)

```
static void gpio_callback_func(pin_t pin, uintptr_t param) {
    //……
    inputValue = uapi_gpio_get_val(SW2_GPIO);
    if(inputValue == GPIO_LEVEL_LOW) {
        uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_HIGH);
    } else {
        uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);
    }
}
static void *button_task(const char *arg) {
    //……                                                    //初始化GPIO、按键和LED初始态等，略
    errcode_t ret = uapi_gpio_register_isr_func(SW2_GPIO, GPIO_INTERRUPT_DEDGE, gpio_callback_func);
    if (ret != 0) {    uapi_gpio_unregister_isr_func(SW2_GPIO);    }
    while (1) {
        uapi_watchdog_kick();                                 //看门狗，后面会讲
        osal_msleep(2000);                                    //延时，模拟其他任务占用CPU
    }
}
```

进行了对硬件"控制"操作
在其他任务中，应避免对同一硬件的控制操作，以防止硬件运行故障
可以使用互斥锁保证LED控制的唯一性

[3.2.2] 中断的定义和应用

- 多个中断事件带来的问题

```
static void gpio_callback_func(pin_t pin, uintptr_t param) {
    //……
    inputValue = uapi_gpio_get_val(SW2_GPIO);
    if(inputValue == GPIO_LEVEL_LOW) {
        uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_HIGH);
    } else {
        uapi_gpio_set_val(LED_GPIO, GPIO_LEVEL_LOW);
    }
}
static void *button_task(const char *arg) {
    //……
    errcode_t ret = uapi_gpio_register_isr_func(SW2_GPIO, GPIO_INTERRUPT_DEDGE, gpio_callback_func);
    if (ret != 0) {    uapi_gpio_unregister_isr_func(SW2_GPIO);    }
    while (1) {
        uapi_watchdog_kick();
        osal_msleep(2000);
    }
}
```

进行了对硬件"控制"操作
在其他任务中，应避免对同一硬件的控制操作，以防止硬件运行故障
可以使用互斥锁保证资源控制的唯一性

//初始化GPIO、按键和LED初始态等，略

此处：只有一个按键，只开了一个中断
Q：如果有两个按键怎么办

连在一起？额外芯片？加个中断？

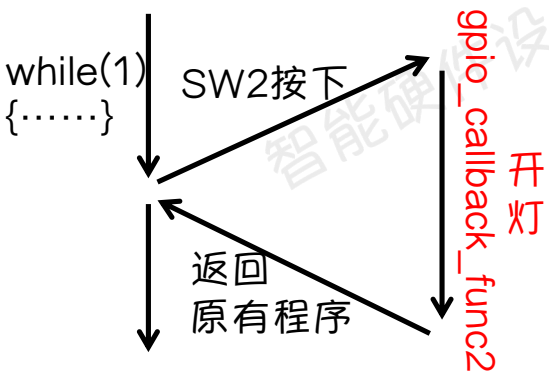
3.2 中断系统

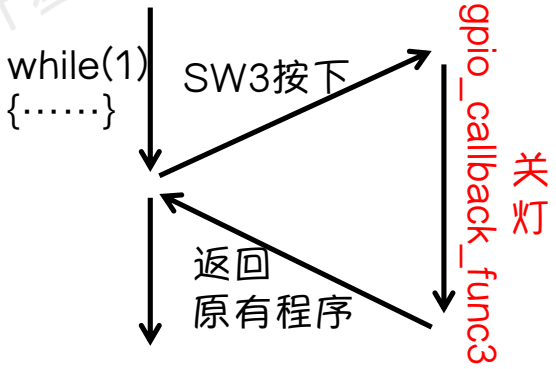
[3.2.2] 中断的定义和应用

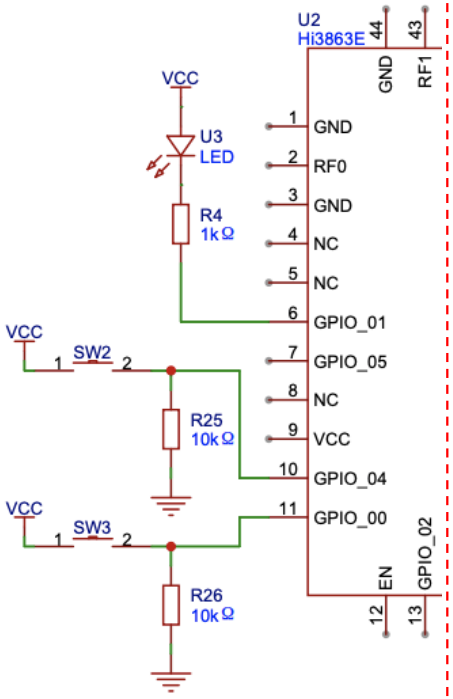
- 多个中断事件带来的问题
  - 多个中断事件(中断源)，且中断事件可以被区分
    - SW2按键开灯，SW3按键关灯

```
errcode_t ret = uapi_gpio_register_isr_func(SW2_GPIO,
                GPIO_INTERRUPT_RISING_EDGE, gpio_callback_func2);
```

```
errcode_t ret = uapi_gpio_register_isr_func(SW3_GPIO,
                GPIO_INTERRUPT_RISING_EDGE, gpio_callback_func3);
```





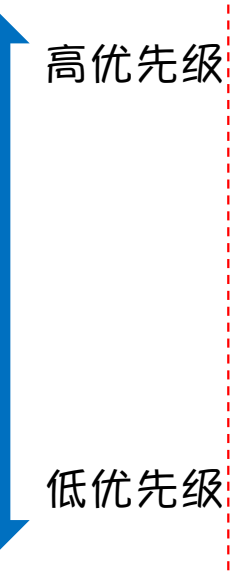


如果在执行gpio_callback_func2()时，产生了SW3按键中断事件会怎么样

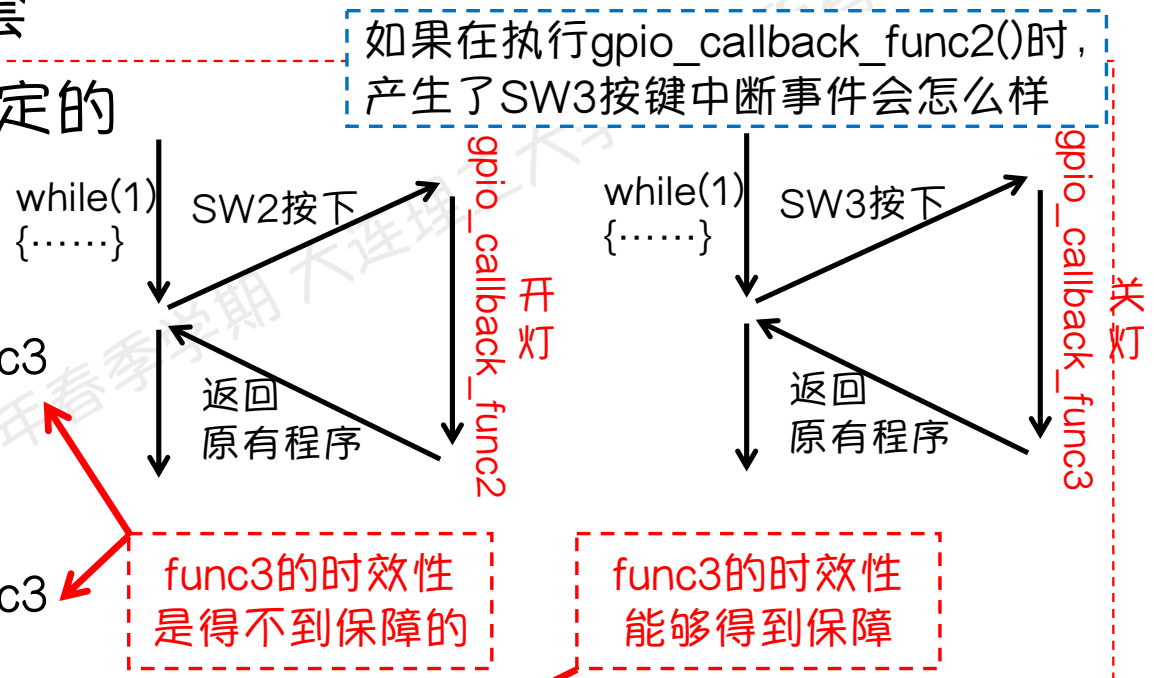
3.2 中断系统

[3.2.3] 中断的优先级和嵌套

- 中断优先级：中断事件被处理的顺序机制
  - 多个中断同时发生时，优先级决定了哪个中断会被处理器优先响应
  - 中断优先级用于确保关键性或时间敏感的任务能够得到及时处理
    - 每个中断都需要分配(或使用默认的)优先级
    - 优先级数值(Prio(n))越小，优先级越高
      - 目前绝大部分MCU/SoC遵守该规则
      - 优先级为0的中断的优先级最高
    - 高优先级的中断可以打断低优先级的中断服务程序
      - 低优先级的中断不能打断高优先级的中断服务程序
      - 相同优先级的中断不能打断同优先级的中断服务程序
  - 中断涉及到现场保护等机制，每次处理中断都要占用系统资源

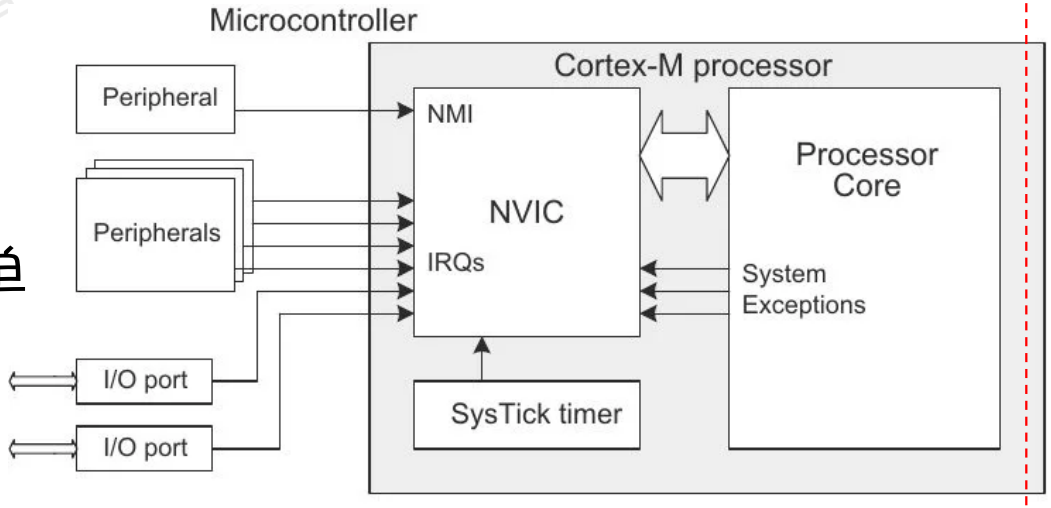


- [3.2.3] 中断的优先级和嵌套
  - 中断能否嵌套 - 优先级决定的
    - Prio(SW2) < Prio(SW3)
      - 不能打断，不构成嵌套
      - fucn2执行完，再执行func3
    - Prio(SW2) = Prio(SW3)
      - 不能打断，不构成嵌套
      - fucn2执行完，再执行func3
    - Prio(SW2) > Prio(SW3)
      - fucn2能够被打断，构成中断嵌套
      - fucn2执行过程中，执行func3，func3执行完，再继续执行func2



中断的嵌套机制在中断优先级的调控下，可以按照用户意图，保障特定事件的时效性

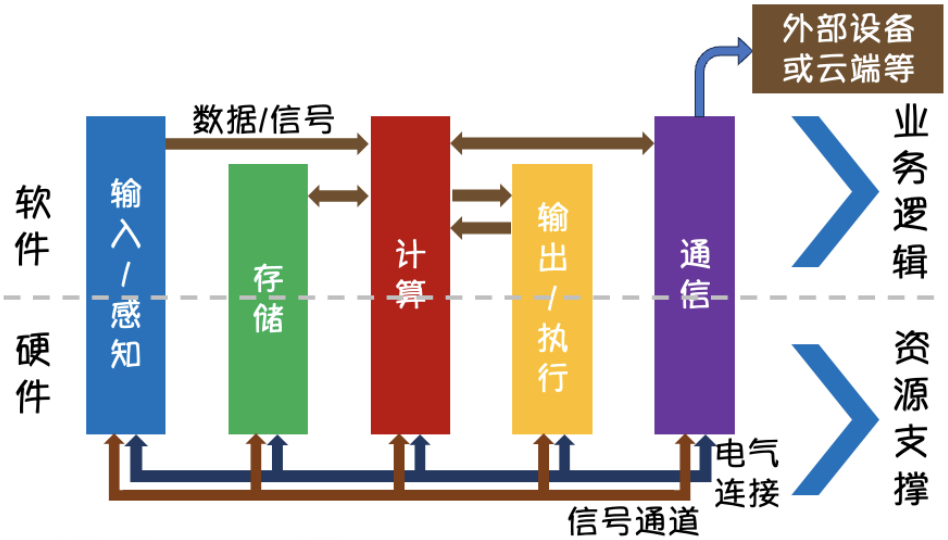
- [3.2.3] 中断的优先级和嵌套
  - 常规MCU/SoC的中断系统结构(Cortex-M为例)
    - 使用NVIC(Nested Vectored Interrupt Controller)管理中断相关功能
      - 中断向量表：存储中断源对应的中断服务程序的地址
      - 中断优先级控制：为不同的中断源设置优先级
      - 中断的使能与禁用
      - 中断的嵌套
      - 中断标志位及处理
  - 星闪SoC WS63的中断较简单
    - 目前尚未见完善的中断资料
    - 在后续课程中，会使用到中断



无法获取其他SoC公开资料

3.3 模拟数字转换系统

- [3.3.1] 现代信息系统中的模拟信号
  - 以数字信号为主的现代信息系统(包括智能硬件系统)
    - 但是现实世界中，模拟信号居多
      - 温度、湿度
      - 电压、电流
      - 气压、水压
      - 光照强度、屏幕亮度
      - 声音大小、信号强度



这是多少摄氏度

3.3 模拟数字转换系统

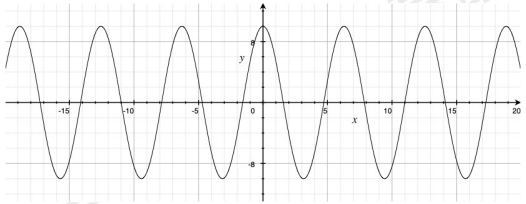
●[3.3.1] 现代信息系统中的模拟信号

- 以数字信号为主的现代信息系统(包括智能硬件系统)
  - 模拟信号的优点：
    - 时间连续性：时间是连续的，没有间隔
    - 数值连续性：信号在某个范围内连续变化
    - 分辨率无限：可以无限精确的反应变化
    - 表达能力强：真实还原连续变化的物理量

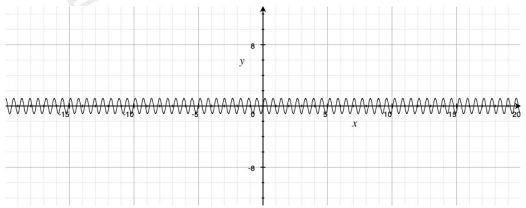
缺点同样明显
传输系统复杂
容易受到干扰
不易精确复制
不具有容错性

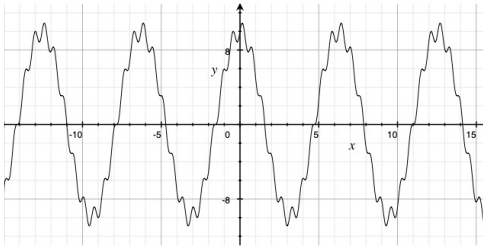
存储
传输
复制
都会
加剧
失真

正常信号
y=10cos(x)

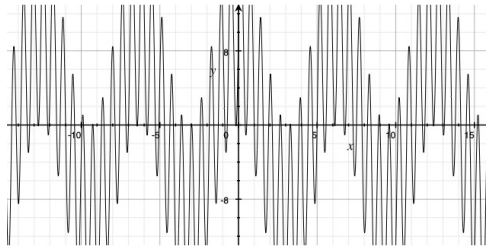


噪声信号
y=sin(10x)





y=10cos(x)+sin(10x)



y=10cos(x)+10sin(10x)

实际的噪声信号无法表达成数学形式，也就无法去除

3.3 模拟数字转换系统

●[3.3.2] 模数转换器功能与主要指标

- 以数字信号为主的现代信息系统(包括智能硬件系统)
  - 在输入/感知阶段就首先将模拟信号转换为数字信号再进行后续的处理
  - 模拟到数字转换的外设：ADC(Analog-to-Digital Converter，模数转换器)
  - ADC的关键指标

读温度的流程：聚焦+拍照（采样） → 看照片（保持） → 读数值（量化） → 写下来（编码）

读取到的数值：

- 数值的位数：37.7度、37.76度、37.766度、37.7666度、…… —— 分辨率
- 读取的速度：多久能完成一次完整的过程，一秒完成多少次转换 —— 转换速率
- 数值的范围：34.5度-42.4度，或者其他类型温度计的范围 —— 量程
- 结果的形式：十进制、二进制、八进制、十六进制？电信号、光信号？ —— 输出形式

3.3 模拟数字转换系统

[3.3.2] 模数转换器功能与主要指标

- ADC的关键参数：分辨率(Resolution)

数值的位数：37.7度、37.76度、37.766度、37.7666度、……　**位数的多少**

  - ADC将模拟信号量化为离散数字信号的二进制位数，通常以位(bit)表示
    - 表征：ADC的量化精度，分辨率越高对模拟信号的表达越好
    - 分辨率越高，精确程度越高
    - 分辨率越高，存储空间越大

8位　-> 2^8=256个量化的等级

10位　-> 2^{10}=1024个量化的等级

10位　-> 2^{12}=4096个量化的等级

    - 常见的MCU/SoC的ADC分辨率
      - STM32H757：ADC，16位
      - ESP32-C2：ADC，12位
      - WS63：ADC，12位

3.3 模拟数字转换系统

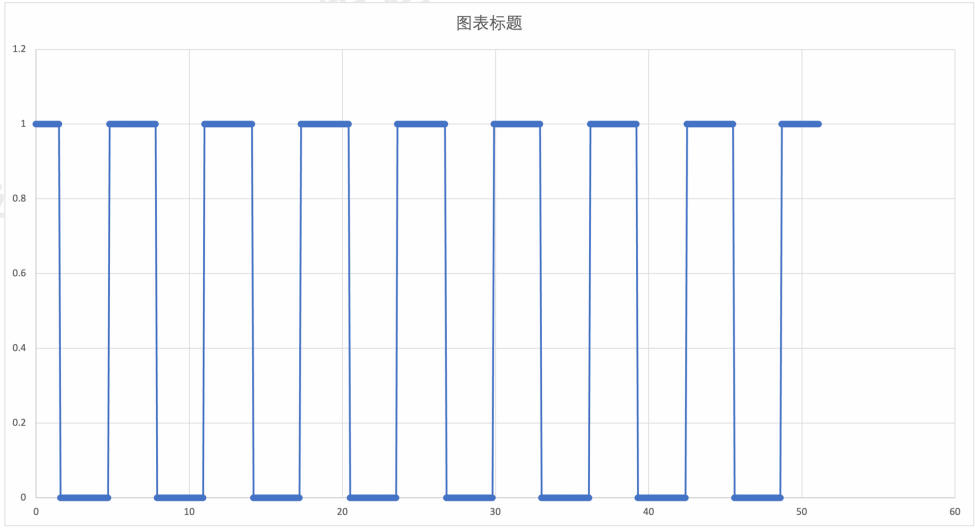
[3.3.2] 模数转换器功能与主要指标

- ADC的关键参数：转换速率(Conversion Rate)
  - ADC每秒钟能够完成的模数转换的次数，通常以SPS为单位
    - 转换速率越快，单位时间采集的数据越多，对模拟信号的表达越好
    - 转换速率越快，精确程度越高；转换速率越快，存储空间越大

对于信号
y=10cosx+10

横轴：转换速率相关
对模拟信号在时间精细度方面的程度

纵轴：分辨率相关
对模拟信号在幅值精细度方面的程度

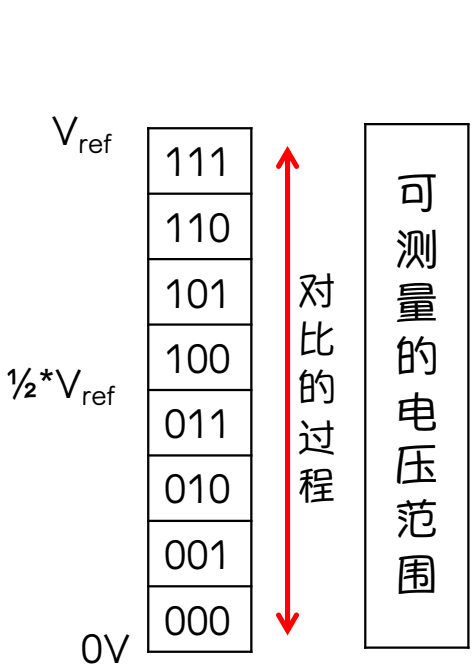


0.25 SPS
0.5 SPS
1 SPS
2 SPS
3.3 SPS
10 SPS

1 bit
2 bits
3 bits
4 bits
8bits

[3.3.2] 模数转换器功能与主要指标

- ADC的关键参数：量程(Full-Scale Range)
  - ADC能够测量的输入信号的电压范围，对于N位分辨率的ADC而言
    - 量程的最小值：输出数值为 0
    - 量程的最大值：输出数值为 2^N-1
  - 超出该范围的电压：无法测量，甚至损坏ADC
  - 数值上量程等于ADC的"参考电压V_{ref}"(共GND)
- ADC的关键参数：输出形式(Output)
  - ADC以何种形式将转换后的结果输出
    - 信息的形式：电信号、光信号、电磁波、其他形式
    - 信息的格式：十进制、十六进制
    - 通信的协议：以太网、Wi-Fi、BT、SLE、I^2C、UART



3.3 模拟数字转换系统

- [3.3.2] 模数转换器功能与主要指标
  - ADC的计算方式：分辨率为N
    - 量程范围：[0, V_{ref}]
    - 单端ADC(共GND)输出值：$\frac{V_{in}}{V_{ref}} \times (2^N - 1)$
    - 假设某ADC分辨率N=12，V_{ref}=3.3V
      - 最低位的1位代表的电压值(LSB)：$\frac{3.3V}{4096} \approx 0.0008V$
      - 输入0V的转换结果为：0
      - 输入1.65V的转换结果为：$\frac{1.65}{3.3} \times (2^{12} - 1) = 2047$
      - 输入3.3V的转换结果为：4095
      - 如果V_{ref}=5.0V，输入1.65V对应的转换结果为：$\frac{1.65}{5.0} \times (2^{12} - 1) \approx 1351$

3.3 模拟数字转换系统

- [3.3.3] WS63 ADC的主要特性
  - ADC的工作流程总结

ADC工作流程 → 采样 → 保持 → 量化 → 编码

  - ADC的主要特性：ADC是一种低速设备
    - 分辨率为固定的12位，工作频率有15KHz至500KHz等四种可选
    - 采样平均值滤波处理：直接采样(不平均)、2次平均、4次平均、8次平均
      - 优：降低噪声影响、提高测量精度、实现简单；缺：动态响应变慢
    - 多通道采样：6个通道共享1个ADC模块，可设置采样通道进行轮流采样
      - 优：单一ADC模块实现多个输入通道的轮流转换；缺：采样出现滞后
    - 多种扫描模式：自动扫描(单一)转换，手动启动转换
    - FIFO功能：FIFO缓存，FIFO水线中断和满中断，FIFO状态查询等

3.3 模拟数字转换系统

[3.3.3] WS63 ADC的主要特性

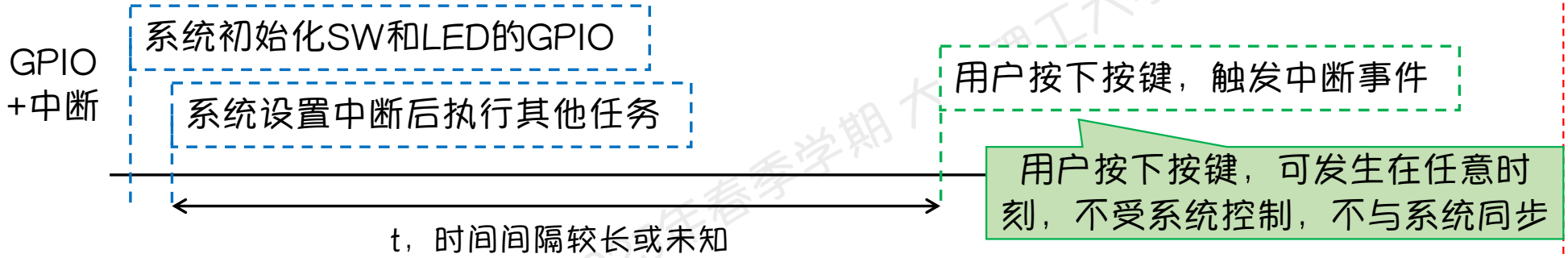
- ADC工作方式的讨论：ADC是一种低速外设
  - 低速外设：相对于CPU而言传输速度低、处理时间长的外设模块，主要用于处理简单的输入/输出任务，或低速数据通信任务
    - 所有连接在APB总线上的外设都属于低速外设
      - GPIO、ADC、I2C、UART、SPI、RTC、Timer、PWM、CAN等
    - 特点：低速率、低功耗、实现简单、实时性要求低、高可靠性
  - 访问低速外设的特点
    - 回顾一下GPIO的SW-LED的学习过程
      - 轮询检测按键状态 > 耗费大量时间等待用户按键，不能执行其他任务
      - 设置中断模式检测 > 设置中断、等待触发，中断服务程序中处理事件

中断可以有效避免处理器等待用户操作导致的系统性能下降(已经讲过)

3.3 模拟数字转换系统

[3.3.3] WS63 ADC的主要特性

- ADC工作方式的讨论：异步机制访问

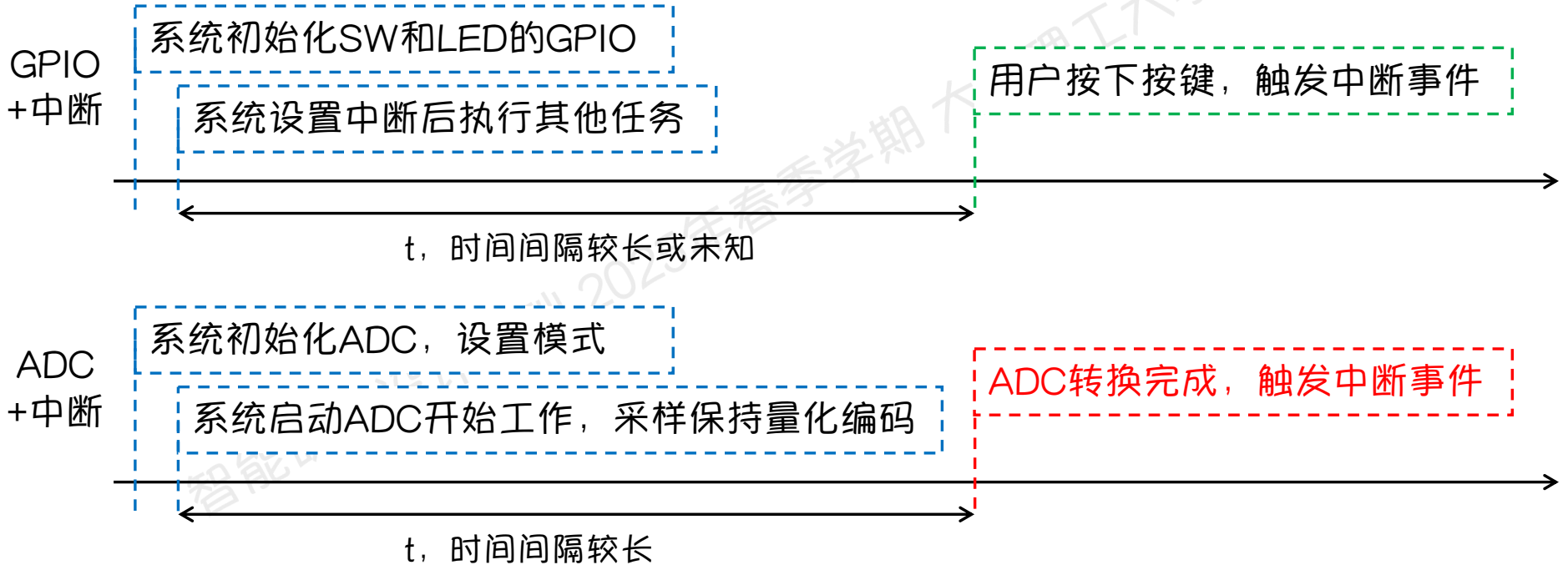


- 同步机制：多任务按照特定顺序或规则执行，保证资源和数据的一致性
- 异步机制：系统运行过程中，任务的处理无需等待前一任务完成，通过中断等方式，允许多个任务并发或交错进行，提高系统效率和响应能力
  - 特点1：非阻塞执行，任务发起请求后立刻范围，不需要等待结果输出
  - 特点2：事件驱动，任务的完成由中断触发或事件通知，非主动轮询
  - 特点3：并发处理，同时处理多个任务，提升资源利用率
  - 特点4：延迟处理，任务启动后不立刻返回结果，需要延迟处理结果数据

3.3 模拟数字转换系统

[3.3.3] WS63 ADC的主要特性

- ADC工作方式的讨论：异步机制访问



异步机制广泛的应用在智能硬件系统、网络编程、GUI开发、数据采集和多任务系统中

[3.3.4] WS63 ADC的工作方式

WS63 ADC的主要API

①：errcode_t uapi_adc_init(adc_clock_t clock)

②：errcode_t uapi_adc_deinit(void)

③：void uapi_adc_power_en(afe_scan_mode_t afe_scan_mode, bool en)

④：errcode_t uapi_adc_open_channel(uint8_t channel)

⑤：errcode_t uapi_adc_close_channel(uint8_t channel)

⑥：int32_t uapi_adc_manual_sample(uint8_t channel)

⑦：errcode_t uapi_adc_auto_scan_ch_enable(uint8_t channel, adc_scan_config_t config, adc_callback_t callback)

⑧：errcode_t uapi_adc_auto_scan_ch_disable(uint8_t channel)

⑨：void uapi_adc_auto_scan_disable(void)

A 禁止指定通道的多次自动转换

B 启动多次自动转换

C 初始化ADC

E 去初始化ADC

F 启动ADC

G 打开指定ADC通道

F 关闭指定ADC通道

D 人工触发单次转换

H 禁止全部自动转换

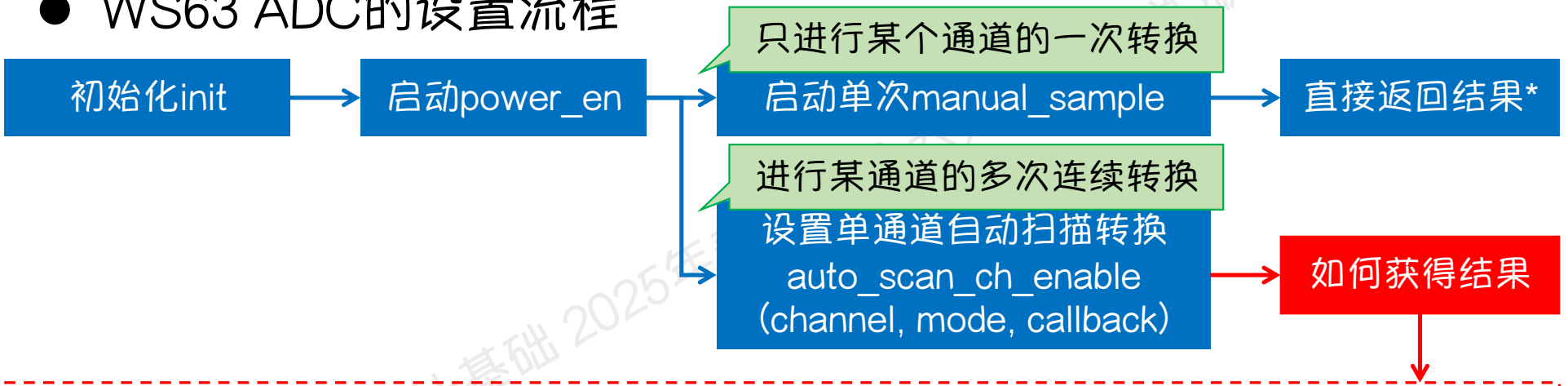
3.3 模拟数字转换系统

- [3.3.4] WS63 ADC的工作方式
  - WS63 ADC的主要API
    - 初始化或去初始化：init、deinit
    - 启动(模块上电)：power_en
    - 打开或关闭指定通道：open、close
    - 启动单通道自动扫描转换：auto_scan_ch_enable
    - 禁止自动扫描转换：auto_scan_ch_disable、auto_scan_disable
    - 启动单次转换：manual_sample(返回值类型int_32)
    - 与ADC状态相关
      - 是否工作：is_using
      - 是否自动扫描转换：is_enabled

3.3 模拟数字转换系统

[3.3.4] WS63 ADC的工作方式

WS63 ADC的设置流程



ADC的主要API⑩：void(*adc_callback_t)(uint8_t channel, uint32_t *buffer, uint32_t length, bool *next)

- uint8_t channel：触发当前中断的通道，0至5
- uint32_t *buffer：自动扫描采样转换结果存放位置
- uint32_t length：数据的长度
- bool *next：继续自动转换或停止

ADC自动扫描转换并将数据保存到FIFO会占用大量时间，使用异步机制可以提升系统效率

目前WS63 ADC SDK还有超级多的BUG

3.4 本章作业

- [3.4.0] 作业与思考

1. GPIO有哪些属性(功能)是需要控制才能使用的
2. 灌电流和拉电流是在什么情况下才能实现
3. 中断在系统中有何应用意义
4. 中断服务程序在中断中发挥了何种作用，有何注意事项
5. 简述中断优先级的作用
6. 简述ADC的分辨率和量程的含义
7. 异步工作机制有哪些优点
8. 思考继续：SoC如何通过外设与外界进行信息交互(数据传输)

GPIO	UART	I2C	LSADC	PWM
Wi-Fi、RADAR、SLE、BLE		Timer	WDT	TCXO