AI8051U 库函数使用说明

更新日期：2025 年 3 月 20 日

目录

I/O 部分

主要功能

设置 I/O 口的模式

主要函数

```
void set_io_mode(io_mode mode, io_name Pinx, ..., Pin_End);
//批量设置 I/O 的模式
```

函数输入参数解释

无返回值 set_io_mode(需要设置的 I/O 模式，
需要设置的引脚编号 1(因为头文件已经定义了 P00 这种，所以库函数使用 Pin00 代替)，
需要设置的引脚编号 2…(Pinxy 中，x 范围 0~5，y 范围 0~7)，
引脚编号结束标志(固定为 Pin_End));

参数配置表

I/O 模式枚举	模式名字	详细说明	默认状态
pu_mode	准双向口模式 pull-up	当 I/O 口未连接时，通过内部的弱上拉电阻将其电平拉高。适用于需要默认高电平的场合，如按键输入。	高阻
注意事项：准双向口模式适用于需要同时输入和输出的场合，但在输出低电平时，外部信号不应驱动高电平，否则可能导致电流过大。			
pp_mode	推挽输出模式 push-pull	适用于需要强驱动能力的场合，如驱动 LED 或继电器。	
注意事项：推挽输出不能直接用于线与逻辑，可能导致短路。			
hz_mode	高阻输入模式 high-z	I/O 口呈现高阻抗状态，不驱动外部电路。适用于需要读取外部信号的场合，如模拟信号输入。	
注意事项：高阻状态下易受干扰，需注意信号屏蔽和滤波。			
od_mode	开漏模式 open-drain	输出低电平时，I/O 口和地等电位；输出高电平时，I/O 口呈高阻态，需外部上拉电阻。适用于线与逻辑或多设备共享总线（如 I2C）。	
注意事项：需外接上拉电阻，电阻值需根据总线速度和负载计算。			
dis_pur en_pur	上拉电阻配置 pull-up-resistor	配置 I/O 口内部上拉电阻的启用，阻值约为 4K。适用于需要默认高电平的场合。	关闭
注意事项：启用上拉电阻会增加功耗，需根据实际需求选择是否启用。			
dis_pdr en_pdr	下拉电阻配置 pull-down-resistor	配置 I/O 口内部下拉电阻的启用，阻值约为 47K。适用于需要默认低电平的场合。	关闭
注意事项：启用下拉电阻会增加功耗，需根据实际需求选择是否启用。			
en_schmitt_trig dis_schmitt_trig	配置施密特触发 schmitt trigger	施密特触发器具有滞回特性，用于消除输入信号的抖动。适用于需要稳定输入信号的场合，如按键或传感器输入。	开启
注意事项：禁用施密特触发可能导致信号抖动，影响系统稳定性。			
low_speed high_speed	电平转换速度 transition speed	低速模式下电平转换较慢，上下冲较小；高速模式下电平转换较快，但上下冲较大。适用于不同信号频率的场合。	低速
注意事项：高速模式下可能产生电磁干扰，需注意信号完整性设计。			

small_current big_current	驱动电流大小 drive current	小电流模式下驱动能力较弱，适用于一般使用场合；大电流模式下驱动能力较强，适用于驱动大负载。	小电流
注意事项：大电流模式下功耗较高，需注意电源设计和 I/O 承载能力。			
en_dinput dis_dinput	配置数字输入 digital input	启用数字输入功能时，MCU 可读取外部端口的电平；禁用时，I/O 口仅作为输出或模拟输入。适用于需要切换数字输入功能的场合。	开启
注意事项：进入低功耗模式前需禁用数字输入功能，否则可能导致额外功耗。			
dis_auto_config en_auto_config	自动配置模式 auto configuration	启用自动配置模式时，外设模块自动配置 I/O 口模式；禁用时，需手动配置 PxM0/PxM1 寄存器。适用于需要灵活控制 I/O 模式的场合。	关闭
注意事项：自动配置模式可能覆盖手动配置，需根据外设需求选择是否启用。			

使用举例

示例 1：设置单个引脚为推挽输出模式

场景：将 P00 引脚设置为推挽输出模式，用于驱动 LED。

代码：

```
set_io_mode(pp_mode, Pin00, Pin_End);
```

说明：

1. pp_mode 表示推挽输出模式。
2. Pin00 表示 P00 引脚。
3. Pin_End 表示参数列表结束。

示例 2：设置多个引脚为高阻输入模式

场景：将 P10、P21、P32 引脚设置为高阻输入模式，用于读取外部传感器的信号。

代码：

```
set_io_mode(hz_mode, Pin10, Pin21, Pin32, Pin_End);
```

说明：

1. hz_mode 表示高阻输入模式。
2. Pin10、Pin21、Pin32 表示需要设置的引脚。
3. Pin_End 表示参数列表结束。

示例 3：设置多个引脚为开漏输出模式

场景：将 P01、P12、P23 引脚设置为开漏输出模式，用于 I2C 总线通信。

代码：

```
set_io_mode(od_mode, Pin01, Pin12, Pin23, Pin_End);
```

说明：

1. od_mode 表示开漏输出模式。
2. Pin01、Pin12、Pin23 表示需要设置的引脚。
3. Pin_End 表示参数列表结束。

示例 4：设置多个引脚为上拉输入模式

场景：将 P02、P13、P24 引脚设置为上拉输入模式，用于按键检测。

代码：

```
set_io_mode(pu_mode, Pin02, Pin13, Pin24, Pin_End);
```

说明：

1. pu_mode 表示上拉输入模式。
2. Pin02、Pin13、Pin24 表示需要设置的引脚。
3. Pin_End 表示参数列表结束。

示例 5：设置多个引脚为高速推挽输出模式

场景：将 P03、P14、P25 引脚设置为高速推挽输出模式，用于驱动高速信号（如 PWM 输出）。

代码：

```
set_io_mode(pp_mode, Pin03, Pin14, Pin25, Pin_End);
set_io_mode(high_speed, Pin03, Pin14, Pin25, Pin_End);
```

说明：

1. pp_mode 表示推挽输出模式。
2. high_speed 表示高速电平转换模式。
3. Pin03、Pin14、Pin25 表示需要设置的引脚。
4. Pin_End 表示参数列表结束。

示例 6：设置多个引脚为低功耗模式

场景：将 P04、P15、P26 引脚设置为低功耗模式，关闭数字输入功能以降低功耗。

代码：

```
set_io_mode(dis_dinput, Pin04, Pin15, Pin26, Pin_End);
```

说明：

1. dis_dinput 表示关闭数字输入功能。
2. Pin04、Pin15、Pin26 表示需要设置的引脚。
3. Pin_End 表示参数列表结束。

示例 7：设置多个引脚为自动配置模式

场景：将 P05、P16、P27 引脚设置为自动配置模式，由外设模块自动配置 I/O 模式。

代码：

```
set_io_mode(en_auto_config, Pin05, Pin16, Pin27, Pin_End);
```

说明：

1. en_auto_config 表示启用自动配置模式。
2. Pin05、Pin16、Pin27 表示需要设置的引脚。
3. Pin_End 表示参数列表结束。

示例 8：混合设置多个引脚的不同模式

场景：将 P00 设置为推挽输出模式，P11 设置为高阻输入模式，P22 设置为开漏输出模式。

代码：

```
set_io_mode(pp_mode, Pin00, Pin_End);
set_io_mode(hz_mode, Pin11, Pin_End);
set_io_mode(od_mode, Pin22, Pin_End);
```

说明：

1. 分别调用 set_io_mode 函数设置不同引脚的模式。
2. 每个设置都以 Pin_End 结束。

详细解释

函数使用注意事项

mode 为 I/O 的模式，后面为可变参数的 io_name 参数。

需要注意的是，输入完后需要添加 Pin_End 作为结束符

例如 set_io_mode(pu_mode,Pin00,Pin21,Pin32,Pin_End);

就是将 P00,P21,P32 这 3 个 I/O 设置为上拉输入模式

io_name 参数为 Pinxy 格式，其中 x 范围是 0~5，y 范围是 0~7。

为什么使用 Pin00 这种格式？

是因为官方的头文件已经使用了 P00 这种格式的定义，来表示实际的 I/O 引脚，所以这里为了方式冲突，就使用了 Pin00 这种格式。

普通 I/O 口均可中断部分

主要功能

普通 I/O 口均可中断部分，可以在任意 I/O 上

需要注意的是，这个函数需要依赖 set_io.h，否则无法使用。

主要函数

```
void set_ioint_mode(ioint_mode, io_name pin, ..., Pin_End);
//批量设置 I/O 中断的模式
char get_ioint_state(io_name pin);
//获取对应引脚是否产生中断
void set_ioint_isr(io_name pin, void (*isr)(void));
//设置中断服务程序
```

参数配置表

I/O 模式枚举	模式名字	简单说明	默认状态
en_int	使能 I/O 中断	使能 I/O 口的中断功能。	禁止
dis_int	禁止 I/O 中断	禁止 I/O 口的中断功能。	
falling_edge_mode	下降沿中断模式	当检测到下降沿时触发中断	下降沿
rising_edge_mode	上升沿中断模式	当检测到上升沿时触发中断。	
low_level_mode	低电平中断模式	当检测到低电平时触发中断。	
high_level_mode	高电平中断模式	当检测到高电平时触发中断。	
priority_base	中断优先级最低(0)	设置中断优先级为最低（默认）。	最低优先级 0
priority_low	中断优先级低(1)	设置中断优先级为低。	
priority_medium	中断优先级中(2)	设置中断优先级为中等。	
priority_high	中断优先级高(3)	设置中断优先级为高。	
en_wakeup	使能 I/O 唤醒	使能 I/O 口的唤醒功能。	禁止
dis_wakeup	禁止 I/O 唤醒	禁止 I/O 口的唤醒功能。	

函数输入参数解释

无返回值 set_ioint_mode(需要设置的 I/O 中断模式，

需要设置的引脚编号 1(因为头文件已经定义了 P00 这种，所以库函数使用 Pin00 代替)，

需要设置的引脚编号 2…(Pinxy 中，x 范围 0~5，y 范围 0~7)，

引脚编号结束标志(固定为 Pin_End));

返回值(0、1) get_ioint_state(需要设置的引脚编号 1(因为头文件已经定义了 P00 这种，所以库函数使用 Pin00 代替));

通常搭配 if 直接使用，例如 if(get_ioint_state(Pin00)){/*执行内容*/};

使用举例

示例 1：使能多个引脚的中断功能

场景：使能 P00、P15、P23 的中断功能，用于检测下降沿触发的事件（如按键按下）。

代码：

```
set_ioint_mode(en_int, Pin00, Pin15, Pin23, Pin_End);
```

说明：

1.en_int 表示使能 I/O 中断功能。

2.Pin00、Pin15、Pin23 表示需要使能中断的引脚。

3.Pin_End 表示参数列表结束。

未显式设置中断模式时，默认采用下降沿触发方式。

示例 2：设置上升沿触发模式

场景：将 P10、P21 设置为上升沿触发模式，用于检测信号上升沿（如传感器信号）。

代码：

```
set_ioint_mode(rising_edge_mode, Pin10, Pin21, Pin_End);
```

说明：

1.rising_edge_mode 表示设置为上升沿触发模式。

2.Pin10、Pin21 为目标引脚，设置后需额外调用 en_int 使能中断。

3.多个触发模式需分开设置，例如电平触发与边沿触发不可混用。

示例 3：组合优先级设置与中断使能

场景：设置 P05 为高优先级中断，用于快速响应紧急事件。

代码：

```
set_ioint_mode(priority_high, Pin05, Pin_End); // 先设置优先级
set_ioint_mode(en_int, Pin05, Pin_End); // 再使能中断
```

说明：

1.priority_high 设置中断优先级为最高级别。

2.en_int 需在优先级设置后调用以生效。

3.优先级配置需在中断使能前完成。

示例 4：检测中断状态并处理

场景：在主循环中轮询 P00 和 P11 的中断状态，触发后执行对应操作。

代码：

```
if (get_ioint_state(Pin00))
{handle_button_press(); // P00 中断处理（如按键按下）}
if (get_ioint_state(Pin11))
{read_sensor_data(); // P11 中断处理（如传感器信号）}
```

说明：

1.get_ioint_state(Pin00) 返回 1 表示检测到中断，查询后标志位自动清除。

2.需确保引脚已通过 set_ioint_mode 配置中断模式和使能。

3.适用于非阻塞式实时检测场景。

示例 5：使能低电平唤醒功能

场景：配置 P32 引脚用于低电平唤醒设备，实现低功耗模式下的外部唤醒。
代码：

```
    set_ioint_mode(low_level_mode, Pin32, Pin_End); // 设为低电平触发
       set_ioint_mode(en_wakeup, Pin32, Pin_End); // 使能唤醒功能
```

说明：
1.low_level_mode 设置低电平触发模式。
2.en_wakeup 使能唤醒功能，需与触发模式配合使用。
3.唤醒功能通常用于睡眠/待机模式恢复。

示例 6：批量禁止中断功能
场景：系统初始化时禁用 P03、P14 引脚的中断功能。
代码：

```
            set_ioint_mode(dis_int, Pin03, Pin14, Pin_End);
```

说明：
1.dis_int 表示禁止引脚的中断功能，中断默认就是关闭的。
2.适用于开启中断后，需要临时关闭中断或初始化时清理状态的场景。

示例 7：设置 I/O 中断的回调函数
场景：对于响应要求特别高的情况，要求不高建议还是用主循环查询标志位方式
代码：

```
void isr(void){P00 = ~P00;}//定义用户回调函数
void main(void){
    ...
    set_ioint_isr(Pin10, isr);//设置 isr 函数为 P1 的 I/O 中断函数
    //传入 Pin10~Pin17 都是设置 P1 口的中断函数，效果是一样的
    while(1){
        ...
    }
}
```

说明：
1.可以定义任意函数名字，不一定非要是 isr()
2.设置 Pin10~Pin17 都可以设置为 P1 接口的中断函数，后设置的会覆盖之前设置的
3.函数会在 P1 口任意引脚产生中断的时候进行调用
4.后续会支持每个 I/O 均对应一个中断回调函数，目前暂未支持，请等待更新...

详细解释
可以同时设置多个 Pin 脚的模式
第一个参数为 ioint_mode 枚举类型，第二个参数及其后面为 io_name 枚举类型
这是一个变长函数，举一个例子：
set_ioint_mode(en_int,Pin00,Pin01,Pin02,Pin20,Pin_End);
这样是将 Pin00,Pin01,Pin02,Pin20 的 I/O 中断模式都使能，最后必须为 Pin_End
io_name 参数为 PinXx 格式，其中 X 范围是 0~5，x 范围是 0~7。
使用的时候，需要先设置 ioint 的中断模式（上升沿、下降沿之类的），然后再打开 ioint 的中断，就可以使用了。如果不设置中断模式，则默认为下降沿中断模式。

如何获取中断标志？
get_ioint_state 是一个获取中断标志位的函数。
参数为 io_name 枚举类型，返回值为 char 类型，返回值只会出现 0 和 1，0 表示没有中断，1 表示有中断，查询一次后自动清除。
使用上，放在主循环中：if(get_ioint_state(Pin00))//判断 P00 是否有中断

传统的外部中断 INT0~INT4 部分

主要功能

用于设置外部的 INTx 口中断，只能使用固定的引脚

主要函数

```
void set_int_mode(int_mode mode, int_num num, ..., Int_End);
//设置外部 INT0~INT4 的中断触发方式，后续参数为 int_num，详见下表说明。
char get_int_state(int_num num);
//获取 INTx 的中断状态，1 表示有中断，0 表示无中断。
void set_int_isr(int_num num, void (*isr)(void));
// 设置中断服务程序
```

函数输入参数详细解释

无返回值 set_int_mode(需要设置的 INT 模式，

需要设置的 Int 号 1(不同模式对应可设置的范围在下表列出)，

需要设置的 Int 号 2，...，引脚编号结束标志(固定为 Int_End))

注意：Int 需要首字母大写，后续小写，区分于系统定义的 INT0 这样的标志位

参数配置表

INT 模式枚举	模式名称	支持的 INT 号
rising_falling_edge_mode	边沿中断	Int0、Int1
falling_edge_mode	下降沿中断	Int0~Int4
dis_int	关闭中断	Int0~Int4

返回值 0 或 1 get_int_state(需要设置的 INT 号(只能填入一个))

使用举例

示例 1：设置 INT0 和 INT3 为下降沿中断

场景：同时设置多个 INT 引脚。

代码：

```
set_int_mode(falling_edge_mode, Int0, Int3, Int_End);
```

说明：

1.falling_edge_mode 表示使能下降沿中断功能。

2.Int0、Int3 表示需要使能中断的位号。

3.Int_End 表示参数列表结束。

其中，不设置状态下，Int0、Int1 默认为边沿中断，Int2~Int4 默认为下降沿中断。

示例 2：设置 INT1 和 INT2 中断关闭

场景：同时关闭多个 INT 中断功能，在已经设置 Int 模式的情况下(设置后自动打开)

代码：

```
set_int_mode(dis_int, Int1, Int2, Int_End);
```

说明：

1.dis_int 表示关闭中断功能。
2.Int1、Int2 表示需要使能中断的位号。
3.Int_End 表示参数列表结束。

示例 3：检测到 INT0 中断，改变 P00 引脚电平
场景：已经设置过 INT 中断模式的情况下，以下代码是 while(1)函数中的
代码：

```
...
While(1){
                    if(get_int_state(Int0))P00 = ~P00;
}
...
```

说明：
1.dis_int 表示关闭中断功能。
2.Int1、Int2 表示需要使能中断的位号。
3.Int_End 表示参数列表结束。

示例 4：设置 INT 的回调函数
场景：对于响应要求特别高的情况，要求不高建议还是用主循环查询标志位方式
代码：

```
void isr(void){P00 = ~P00;}//定义用户回调函数
void main(void){
    ...
    set_int_isr(Int0, isr);//设置 isr 函数为 Int0 的中断函数
    while(1){
        ...
    }
}
```

说明：
1.可以定义任意函数名字，不一定非要是 isr()
2.函数会在 INT0 产生中断的时候进行调用

详细解释
中断部分的代理处理
get_int_state 是一个查询是否存在中断的函数
实际的中断函数已经被定义并且处理了，用户无需关心中断函数部分的处理，只需要使用 get_int_state 来查询是否存在对应的中断请求即可。

定时器部分

主要功能

用于设置定时器的定时时间，以实现固定时间触发的一些任务场景。

主要函数

```
void set_timer_mode(timer_num num, char* set_time, ..., Timer_End);
//设置定时器的各种参数，支持乱序输入和默认值功能
char get_timer_state(timer_num num);
//获取定时器当前的中断状态，内部的标志位是缓存进行的。
void set_timer_fosc(long fosc);
//用于设置定时器部分的时钟源频率，fosc 为时钟频率，单位为 Hz。
//不使用这个函数的情况下，主频是自动获取，用户可以不用关心这个问题。
void set_timer_isr(timer_num num, void (*isr)(void));
// 设置中断服务程序
```

函数输入参数详细解释

无返回值 set_timer_mode(需要设置的定时器，
需要设置的定时长度(不输入默认为 1s)，是否需要打开中断(默认为打开)，
是否需要输出定时器时钟(默认为不输出)，引脚编号结束标志(固定为 Timer_End))；

返回值 0 或 1 get_timer_state(需要设置的定时器)；

参数配置表

函数可输入参数	描述	参数范围	默认值
Timerx	定时器位号，对应不同的定时器	Timer0~Timer4、Timer11	-
“xs”	设置的定时时间，单位是秒，例如 1s	单次定时不超过 5s@40Mhz	1s(1 秒)
”xms”	设置更短的定时时间，单位是毫秒，例如 80ms		
”xhz”	设置的定时周期，单位是赫兹，例如 10hz		
En_Int	打开定时器中断，不打开中断无法使用	-	En_Int
Dis_Int	关闭定时器中断	-	
En_OutClk	打开定时器时钟输出功能	-	Dis_OutClk
Dis_OutClk	关闭定时器时钟输出功能	-	

使用举例

示例 1：设置定时器 0 为全默认值方式

场景：不想设置参数或者设置默认参数符合想要设置的数值。
代码：

```
set_timer_mode(Timer0, Timer_End);
```

说明：
1.无设置下，默认为 1s 定时，打开中断，不进行时钟输出(补充说明：是一种将定时器的定时周期作为方波输出到固定引脚的功能，例如定时 1S，就能在对应定时器的固定引脚上输出 1S 变化一次的波形)
2.Timer0 表示需要设置的定时器。
3.Timer_End 表示参数列表结束。

示例 2：设置定时器 1 的定时时间为 200ms，并且打开定时器的时钟输出功能
场景：设置较短的定时时间，用示波器检测定时时间。
代码：

```
set_timer_mode(Timer1, “200ms”, En_OutClk, Timer_End);
```

说明：
1.Timer1 表示需要设置的定时器。
2.”200ms”为字符串数据，内部的 200ms 为定时时间。
3.En_OutClk 为定时器时钟输出，定时器溢出的时候会反转 P34(T1CLKO)
5.Timer_End 表示参数列表结束。

示例 3：设置定时器 2 的定时时间为 2.5s，并且在第一次定时到达的时候关闭定时器 2
场景：只需要一次定时的情况，较为简单逻辑场景下。
代码：

```
...
                set_timer_mode(Timer2, ”2.5s”, Timer_End);
while(1){
if(get_timer_state(Timer2))
{
    //其他执行代码
                set_timer_mode(Timer2, Dis_Int, Timer_End);
}
...
}
```

说明：
1.Timer2 表示需要设置的定时器。
2.”2.5s”为字符串数据，内部的 2.5s 为定时时间，支持浮点输入。
3.Dis_Int 为关闭定时器中断，关闭后则 if(get_timer_state(Timer2))不会再进入
5.Timer_End 表示参数列表结束。

示例 4：设置定时器 3 的定时时间为 10hz，打开定时器时钟输出，使用乱序输入。
场景：展示不同单位的输入和乱序输入的情况。
代码：

```
set_timer_mode(Timer3, En_OutClk, ”10hz”, Timer_End);
```

说明：

1.Timer3 表示需要设置的定时器。
2.”10hz”为字符串数据，内部的 10hz 为定时循环频率，换算到时间为 100ms。
3.可以输入的单位有 s、ms、hz（都是小写），分别代表秒、毫秒、赫兹的意思
4.En_OutClk 为打开定时器时钟输出，他和定时时间设置参数顺序可以随意排布。
5.Timer_End 表示参数列表结束，必须在最后。

示例 5：设置定时器的回调函数
场景：对于响应要求特别高的情况，要求不高建议还是用主循环查询标志位方式
代码：

```
void isr(void){P00 = ~P00;}//定义用户回调函数
void main(void){
    ...
    set_timer_isr(Timer0, isr);//设置 isr 函数为定时器 0 的中断函数
    while(1){
        ...
    }
}
```

说明：
1.可以定义任意函数名字，不一定非要是 isr()
2.函数会在定时器 0 产生中断的时候进行调用

详细解释

get_timer_state 则是查询对应的定时器是否到了相应的计时时间，这里需要注意的是，因为使用了查询机制的获取方式。所以，如果任务比较多的情况下，这个定时时间可能被其他任务给挤占而造成定时时间变长（实际是因为主循环执行时间过长）。

串口部分

主要功能

设置串口的各种参数，进行串口的收发通讯

串口支持乱序输入和参数默认值。

主要函数

```
void set_uart_mode(uart_name uart, ..., Uart_End);
//设置串口模式，带有默认值和乱序输入功能。
char get_uart_state(uart_name uart);
//获取串口接收标志，为 1 是传入的串口号接收到了数据，为 0 是没有接收到。
void uart_printf(uart_name uart, const char* str, ...);
//普通 printf 功能，可以通过第一个参数指定需要发送的串口号
void uart_printf(uart_name uart, Hex_Mode, char dat);
//发送单个数据，类似直接操作 SBUF
void uart_printf(uart_name uart, Buff_Mode, char* addr, int len);
//发送缓冲区内的数据，可以指定发送的端口号，发送的数据长度，一般用于发送数据包
```

参数配置表

参数说明	可选参数	默认值	备注
串口 1 可切换引脚	Uart1_P30_1	Uart1_P30_1	第一个是 RXD 第二个是 TXD 例如 P30 为 RXD P31 为 TXD
	Uart1_P36_7		
	Uart1_P16_7		
	Uart1_P43_4		
串口 2 可切换引脚	Uart2_P12_3	Uart2_P12_3	
	Uart2_P42_3		
串口 3 可切换引脚	Uart3_P00_1	Uart3_P00_1	
	Uart3_P50_1		
串口 4 可切换引脚	Uart4_P02_3	Uart4_P02_3	
	Uart4_P52_3		
奇偶校验选择	Base_8b	Base_8b	8 位无校验
	Odd_9b		奇校验
	Even_9b		偶校验

使用定时器几	Use_Timer2	Use_Timer2	使用定时器 2
	Use_Timerx		使用对应定时器
设置波特率	“xbps”	“115200bps”	x 为波特率数字
设置超时字节数	“xbyte”	“64byte”	x 为超时字节数
注：所有类似 bps、byte 这些单位尾缀都是全小写 并且这些参数作为字符串，使用的是英文引号			

使用举例

示例 1：设置串口 1 为默认值（默认值 115200bps，8，n，1）

场景：用于快速设置，串口设置为什么都可以，只是想要快速使用串口的情况

代码：

```
set_uart_mode(Uart1, Uart_End);
```

说明：

1.第一个参数 Uart1 代表串口 1

2.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

示例 2：设置串口 2 修改引脚分配到 Uart2_P42_3

场景：需要切换串口的 IO 到其他分组的情况

代码：

```
set_uart_mode(Uart2, Uart2_P42_3, Uart_End);
```

说明：

1.Uart2_P42_3 的意思是切换到属于串口 2 的 P42 和 P43 分组

2.如果不设置 IO 切换，默认 Uart1 是 P30 和 P31，Uart2 是 P12 和 P13，Uart3 是 P00 和 P01，Uart4 是 P02 和 P03。

2.需要注意，使用除了 P30、P31 外的其他 IO 口时，要先设置 IO 口模式，否则无法通讯

3.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

示例 3：设置串口 1 为 115200bps，串口 2 为 9600bps

场景：需要不同串口使用不同波特率的情况

代码：

```
set_uart_mode(Uart1, Use_Timerx, Uart_End);
set_uart_mode(Uart2, “9600bps”, Use_Timerx, Uart_End);
```

说明：

1.Use_Timerx 的意思是串口使用对应的定时器来作为波特率发生器

2.使用了 Use_Timerx 后，串口 1 会占用定时器 1，串口 2 会占用定时器 2，依次类推

3.使用“9600bps”即可直接设置波特率为 9600bps，需要注意最后的 bps 为全小写

4.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

示例 4：设置串口 1 和串口 2 都为 9600bps

场景：需要不同串口使用相同的波特率，且不想损耗其他定时器资源

代码：

```
set_uart_mode(Uart1, “9600bps”, Use_Timer2, Uart_End);
set_uart_mode(Uart2, “9600bps”, Use_Timer2, Uart_End);
```

说明：

1.Use_Timer2 的意思是串口都使用定时器 2 来作为波特率发生器

2.使用了 Use_Timer2 后，串口 1 会占用定时器 2，串口 2 会占用定时器 2，依次类推

3.使用“9600bps”即可直接设置波特率为 9600bps，需要注意最后的 bps 为全小写

4.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

示例 5：设置串口 1 为默认参数，并且发送一个字符串

场景：需要快速进行调试输出的时候

代码：

```
set_uart_mode(Uart1, Uart_End);
uart_printf(Uart1, "hello world!\r\n");
```

说明：

1.uart_printf 的第一个参数是指定发送的串口号

2.后面是和 printf 一样的操作

3.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

示例 6：设置串口 1 为默认参数，并且发送一个 int 类型的变量

场景：需要快速进行调试输出的时候，查看变量值的变化

代码：

```
int cnt;
...
set_uart_mode(Uart1, Uart_End);
uart_printf(Uart1, "cnt:%d\r\n", cnt);
```

说明：

1.uart_printf 的第一个参数是指定发送的串口号

2.%d 代表输出的是有符号整形，%u 是无符号整形，%f 是浮点类型

3.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

示例 7：接收串口 1 发送的数据并解析固定的格式

场景：需要进行变量交互的时候，例如上位机发送了一个“cnt:128”

代码：

```
#include "stdio.h"//使用 sscanf 必须要包含的头文件
...
int cnt;
void main (void){
    ...
    set_uart_mode(Uart1, Uart_End);
    ...
    while(1){
        if (get_uart_state(Uart1)){
            sscanf(_uart1_rx_buff, "cnt:%d", &cnt);
```

```
            uart_printf(Uart1, "cnt:%d\r\n", cnt);//回传解析到的值
        }
    }
}
```

说明：

1.get_uart_state()用于获取是否接收完成

2._uart1_rx_buff 是串口 1 的接收缓冲区，具体可以查看 set_uart.h

3.最后一个 Uart_End 是必须要带的，否则变长函数无法正常解析停止

4.解析的时候一样可以通过%d 转义字符来实现解析，sscanf 需要传入的是变量的地址

详细解释

乱序输入和参数默认值功能解释

引入了乱序参数输入和默认值功能，分别是什么意思呢？乱序输入就是指输入参数的顺序，除了第一个需要默认指定为串口号以外，其他的只需要遵守对应的格式即可随意输入，不用在意参数的顺序，比如说“115200bps”和“32byte”，两个参数谁先谁后并没有区别。内部会依靠参数对应的特征进行识别后设置。

而默认值功能则是，允许设定的时候不给定参数，即按照默认的参数进行设定。

设置串口参数

set_uart_mode 可以将未设置的参数给定默认值。

设置串口模式，默认配置为 115200 波特率，8 位数据位，1 位停止位。

这里的参数为变长参数，可变参数为波特率、超时中断数据位、奇偶校验功能、串口切换引脚，最后需要使用 Uart_End 结束。

超时中断解释

超时中断的作用是对数据自动分包，64byte 就是数据发送结束后间隔 64 个字节的时间（根据波特率），就会自动中断，然后进行数据分包。

举个例子：

```
        set_uart_mode(Uart1, "32byte", "115200bps", Uart_End);
```

这个的意思是设置串口 1 为 115200 波特率，并切换引脚为 P30 和 P31 上（默认引脚），超时中断为 32byte。

乱序输入和参数注意事项

变长参数部分支持乱序输入，波特率和超时中断需要带上单位 bps 和 byte，中间不要有空格。

如果不输入变长参数，例如：

```
                    set_uart_mode(Uart1, Uart_End);
            //则代表 115200 波特率，64byte，P30P31（UART1 下）
```

即不输入的选项拥有默认值，不设置也可以的。

get_uart_state 用于获取串口状态。
返回值为 0 代表串口没有接收到数据，1 代表串口接收到了数据，并且数据触发了超时中断。

uart_printf 函数详细说明
uart_printf 是一个聚合了三种模式的 printf 多功能函数。

普通 printf 用法，内嵌 printf 函数，可以通过第一个参数实现打印串口的选择
printf 自带长度校验和串口忙标志，超过长度会不打印
如想要更改最大长度，请到 set_uart.h 中改变 uart_len 枚举值的详细数值
如果连续调用 printf，在第一个 printf 没有完成发送的情况下，后续的 pritnf 会被丢弃
如果想要知道对应的串口发送是否忙，可以使用 tx_state[Uart1]这样子来查询（这个是串口 1 的）
uart_printf(Uart1, "hello world!\r\n");//输出 hello world
uart_printf(Uart1,Hex_Mode,0x0f);//输出 0x0f 单字节，类似直接给 SBUF 值
uart_printf(Uart1,Buff_Mode,tmp_str,5);//输出字符串 tmp_str，5 个字节

应用说明
因为 Keil 中并没有给出 vsscanf 函数，所以目前实现读取是直接使用 sscanf 进行。
串口在接收到一个包的数据后，会默认在最后一个数据的后一位添加一个’\0’来方便 sscanf 进行使用。同时，也提供了对应串口的本次接收的数据长度，使用 rx_cnt[Uart1]这个数组即可访问，给数组的位号传入对应的串口号即可。
下面是一个简单的读取并解析串口接收的程序：

```
if (get_uart_state(Uart1))
{
        // 注意：使用 sscanf 需要引入 stdio.h
        sscanf(_uart1_rx_buff, "cnt:%d", &cnt_dat);
        // 缓冲区可以查看 set_uart.h 中缓冲区的定义
        // sscanf 用法，第一个参数是缓冲区，第二个参数是格式化字符串
        //第三个参数是变量地址
        uart_printf(Uart1, "send num:%d\r\n", (int)cnt_dat);
        // 串口 1 打印解析到的数据并显示
}
```

程序中使用 sscanf 对串口接收到的数据进行解析，并且将解析的结果返回给上位机。

ADC 部分

主要功能

可以配置自动连续采样或者单次采样任意数量的 ADC 通道

会按照通道号从大到小自动依次采样，设置一次后只需直接获取 ADC 值，无需关心实际采样过程。

同时提供采样完成标志位，方便及时处理数据。

主要函数

```
void set_adc_mode(adc_mode mode, adc_ch ch, ..., Adc_End);
//第一个参数是可以设置为连续采样和单次采样模式，后面是需要设置的通道号，可以任意多个，最后加上 Adc_End 即可。
char get_adc_state(adc_ch ch);
//获取对应的通道是否采样完成，采样完成则返回 1
```

参数配置表

参数说明	可选参数	默认值	备注
ADC 通道号枚举	Adc0_P10	无默认值，必须传入至少一个 ADC 通道号来设置模式	Adc15_1_19V 是测量的内部 1.19V 基准源
	Adc1_P11		
	...		
	Adc15_1_19V		
ADC 模式选择	single_mode	无默认值，必须传入需要设定的模式	单次转换模式
	cycl_mode		连续转换模式
ADC 读取数据	adc_value[x]	-	x 可以为 ADC 通道号
adc_value[ch]的数组数据需要在 get_adc_state(ch)==1 时才能读取 否则可能读取到错误数据			

使用举例

示例 1：循环读取单个 adc 数据

场景：需要读取单个 ADC 端口数据的情况

代码：

```
set_adc_mode(cycl_mode, Adc0_P10, Adc_End);
//设置为循环触发模式，每次完成 adc 转换后自动开始下一次
...
while(1){
```

```
    if(get_adc_state(Adc0_P10)){show_value = adc_value[Adc0_P10];}
    //获取 ADC0 通道是否转换完成,对通道数据进行读取
}
```

说明：

1.设置了单个的 ADC 通道进行转换

2.可以在主循环中查询对应 ADC 通道是否转换完成，然后再通过 adc_value 数组读取

3.设置 adc 模式函数中，最后必须是 Adc_End，否则无法正确识别函数

示例 2：循环读取多个 adc 数据

场景：需要读取多个 ADC 端口数据的情况

代码：

```
set_adc_mode(cycl_mode, Adc0_P10, Adc1_P11, Adc10_P02, Adc_End);
//设置为循环触发模式，每次完成 adc 转换后自动开始下一次
...
while(1){
if(get_adc_state(Adc0_P10)){show_value = adc_value[Adc0_P10];}
//获取 ADC0 通道是否转换完成,对通道数据进行读取
if(get_adc_state(Adc1_P11)){show_value = adc_value[Adc1_P11];}
//获取 ADC1 通道是否转换完成,对通道数据进行读取
if(get_adc_state(Adc10_P02)){show_value = adc_value[Adc10_P02
];}
//获取 ADC10 通道是否转换完成,对通道数据进行读取

}
```

说明：

1.设置了多个的 ADC 通道进行转换

2.可以在主循环中查询对应 ADC 通道是否转换完成，然后再通过 adc_value 数组读取

3.设置 adc 模式函数中，最后必须是 Adc_End，否则无法正确识别函数

示例 3：连续采样多次后停止通道采样

场景：需要读取多个 ADC 端口数据的情况

代码：

```
              set_adc_mode(single_mode, Adc0_P10, Adc_End);
                //设置完单次模式后，再采样一次后就自动停止了
```

说明：

1.想要在几次 adc 读取后中断读取，只需要将对应 adc 端口设置为单次读取

2.设置完单次模式后，再读取一次，对应通道就会停止读取了

详细解释

ADC 查询方式

ADC 部分库函数内置一个查询表，用于记忆哪些 ADC 通道需要查询，以中断模式来按顺序查询。

所以，ADC 部分库函数并不会在查询的时候堵塞其他函数的运行，所有的操作完全是基于 ADC 中断内的连续调用实现的。

ADC 分频系数设置

同时，在 set_adc.h 中，还可以设置 ADC 部分的分频系数：

#define ADC_DIV 8 // ADC 时钟分频，默认为 8(18 分频，计算为(n+1)*2 分频)，基本不用关注

这个 ADC_DIV 的范围是 0~15，设置分频系数越大，ADC 采样时间越长，ADC 的值也会越稳定

I2C 部分

主要功能

用于和外部的 IIC 设备进行通讯，目前仅支持 IIC 主机模式

(从机模式暂时还未编写程序)

拥有独特的指令串操作方式、任务序号、带有默认值和乱序输入的配置函数。

主要函数

```
void set_i2c_mode(i2c_name i2c, ..., I2c_End);
//设置 i2c 的模式，可以设置通讯速率和切换引脚组别
void set_i2c_cmd(i2c_name i2c, int task_num, i2c_cmd cmd, ..., Cmd_End);
//设置一串 IIC 操作指令，可以指定任务号，实现非堵塞操作
char get_i2c_state(i2c_name i2c, int task_num);
//获取对应任务号的任务是否执行完成，完成后返回 1，没完成则返回 0
void set_i2c_fosc(long fosc);
//设置 i2c 部分计算用时钟，单位 Hz，一般来说不用管，使用内部 HIRC 时会自动获取。
```

参数配置表

参数说明	可选参数	默认值	备注
I2C 外设编号	I2c0	-	
I2C 模式选择	I2c_Master	主机模式	主机模式
	I2c_Slave		从机模式 （暂未支持）
I2C 外设使能	I2c_Enable	I2c_Enable	
	I2c_Disable		
引脚切换	I2c_P24_3	I2c_P24_3	
	I2c_P15_4		
	I2c_P32_3		
通讯速率	“xkhz”	“400khz”	khz 需小写
AI8051U 目前只有一个 I2C 外设，但是保留 I2C 外设编号用于拓展			

I2C 可选命令表

I2C 命令	跟随参数	示例	详细解释
Start	无	-	发送起始信号，标明所有操作的开始
Tx_Dat	char	char Dat=0x80; Tx_Dat,Dat	发送数据，SCL 产生 8 个时钟，同时 SDA 按先发送高位数据的顺序发送 8 位数据
Rack	无	-	接收 ACK，SCL 产生 1 个时钟，SDA 输入状态，接收 I2C 从机返回 ACK(0)或 NACK(1)
Rx_Dat	char *	char Dat; Rx_Dat,&Dat	接收数据，SCL 产生 8 个时钟，同时 SDA 按先接收高位数据的顺序接收 8 位数据
Tack	无	-	发送 ACK(0)，SCL 产生 1 个时钟，SDA 输出状态，发送一个 ACK(0)
Tnak	无	-	发送 NACK(1)，SCL 产生 1 个时钟，SDA 输出状态，发送一个 NACK(1)
Stop	无	-	发送停止信号，标明所有操作的结束
S_Tx_Rack	char	char Dat=0x80; S_Tx_Rack,Dat	发送起始信号和数据后接收 ACK，联合操作
Tx_Rack	char	char Dat=0x80; Tx_Rack,Dat	发送数据后接收 ACK，联合操作
Rx_Tack	char *	char Dat; Rx_Tack,&Dat	接收数据后发送 ACK，联合操作
Rx_Tnak	char *	char Dat; Rx_Tnak,&Dat	接收数据后发送 NACK，联合操作
命令最后需要添加 Cmd_End 以代表结束，本表只适用 set_i2c_cmd()			

使用举例

示例 1：配置 IIC 外设通讯速率和引脚

场景：需要修改通讯速率和引脚的情况，默认值为 400khz 和 I2c_P24_3

代码：

```
set_i2c_mode(I2c0, "50khz", I2c_P32_3, I2c_End);
```

说明：
1.50khz 的通讯速度,默认主机模式，引脚切换为 P32、P33
2.结尾必须带有 I2c_End 标识，否则无法识别参数结束

示例 2：使用指令串对 AT24C02 读数据
场景：读 AT24C02 数据示例
代码：

```
set_i2c_cmd(I2c0, 1, S_Tx_Rack, 0xa0, Tx_Rack, 0x00, S_Tx_Rack, 0xa1, Rx_Tnak, &c, Stop, Cmd_End);
//任务序号 1，器件地址 0xa0，写地址 00，重新开始后器件地址 0xa1，读数据
...
while(1){
    if (get_i2c_state(I2c0, 1)){P20 = 0;}//读完数据后执行任务
}
```

说明：
1.设置指令串可以任意长度，但是如果指令个数超过（#define Max_I2c_Cmd 20）定义的最大指令缓存长度时，需要修改 set_i2c.h 中的最大指令缓存长度宏定义
2.指令执行为中断内加载缓冲指令执行方式，不堵塞主函数
3.set_i2c_cmd()函数只会将指令加载到指令缓冲区，然后直接执行下一句，并不会等待执行完成
4.如果想要查询当前的指令串是否执行完成，需要使用 get_i2c_state 查询指令串绑定的任务号，返回 1 为执行完成，返回 0 为执行未完成

详细解释
指令串结构
在 set_i2c_cmd()函数内，
实现了独特的指令串结构，可以通过一个函数一次性定义完一次 IIC 操作所需的所有指令。并且内部是非堵塞结构的，在 IIC 指令执行的过程中可以执行其他任务，并非延时死等。

任务序号方式
同时，内部使用了任务序号方式，可以同时设置多条的指令串分别到不同的任务中，他们会按照任务序号从小到大的顺序依次执行，只需要在后续查询对应任务序号的完成状态即可知道是否执行完成。
这样，就算较慢的 IIC 通讯速率也不会堵塞程序执行了。

快速配置通讯速率
IIC 配置函数同样支持乱序输入和默认值操作，设置直接通过”400khz”这样的字符串来设置 IIC 通讯时的 SCL 时钟速率，十分方便，无需再计算。
注意单位 khz 需要全小写，否则无法识别。

SPI 部分

主要功能

可支持同时 3 路 SPI 并带有 DMA 支持

设置 SPI 模式时支持默认值和乱序输入操作，目前仅支持 SPI 主机模式

（SPI 从机模式的代码还没写）

主要函数

```
void set_spi_mode(spi_name spi, ..., Spi_End);
//设置 SPI 的默认值，拥有众多参数和默认值，包括时钟极性，时钟边缘这一些
char get_spi_state(spi_name spi);
//获取 SPI 是否发送完成，用于和 spi_printf 函数配合使用
void spi_printf(spi_name spi, Hex_Mode, char dat);
//发送单个字节的数据，类似直接操作 SPDAT 寄存器进行发送
void spi_printf(spi_name spi, Buff_Mode, char* p, int len);
//DMA 方式，直接发送对应数组中指定长度的数据，可以通过_spi0_rx_buff，
_spi1_rx_buff，_spi2_rx_buff 来获取对应 SPI 外设的读取数据。
```

参数配置表

参数说明	可选参数	默认值	备注
SPI 外设编号	SPI0	-	
	SPI1	-	占用串口 1，不可同时使用
	SPI2	-	占用串口 2，不可同时使用
SPI 模式选择	Spi_Master	主机模式	主机模式
	Spi_Slave		从机模式（暂未支持）
I2C 外设使能	Spi_Enable	Spi_Enable	
	Spi_Disable		
引脚 交换功能	NoSw_MOSI_MISO	NoSw_MOSI_ MISO	不交换 MOSI 和 MISO 引脚的顺序
	Sw_MOSI_MISO		交换 MOSI 和 MISO 引脚的顺序
引脚切换	Spi_P14_5_6_7	Spi_P14_5_ 6_7	从前往后分别对应 SS-MOSI-MISO-SCLK 所有的 SPI 外设都可切换这四组引脚，但是不可以相同
	Spi_P24_5_6_7		
	Spi_P40_1_2_3		
	Spi_P35_4_3_2		

SPI 时钟 分频设置	Spi_ClkDiv_2	Spi_ClkDiv _16	例如输入 SPI 时钟为 40Mhz 使用 Spi_ClkDiv_16 则最后的 CLK 实际频率则为 40/16=2.5Mhz
	Spi_ClkDiv_4		
	Spi_ClkDiv_8		
	Spi_ClkDiv_16		
数据发送时 的优先顺序	MSB	MSB	先发送/接收数据的高位
	LSB		先发送/接收数据的低位
SPI 时钟极性	Low_Rising	High_Falli ng/Cpol_1	SCLK 空闲时为低电平，SCLK 的前时钟沿为上升沿，后时钟沿为下降沿，两个参数是同样的效果
	Cpol_0		
	High_Falling		SCLK 空闲时为高电平，SCLK 的前时钟沿为下降沿，后时钟沿为上升沿，两个参数是同样的效果
	Cpol_1		
SPI 时钟边沿	Out_In	Out_In/ Cpha_1	数据在 SCLK 的前时钟沿驱动，后时钟沿采样，两个参数是同样的效果
	Cpha_0		
	In_Out		数据在 SCLK 的后时钟沿驱动，前时钟沿采样，两个参数是同样的效果
	Cpha_1		
在 set_spi.h 中，可以通过 spi_len 枚举参数调整 SPI 内部缓冲区大小 可以通过_spi0_rx_buff 读取 SPI0 同步接收到的数据，同理还有 _spi1_rx_buff、_spi2_rx_buff 来接收 SPI1 和 SPI2 的			

使用举例

示例 1：设置 SPI 为默认参数，并且切换引脚

场景：全部设置默认参数，切换 SPI 的 IO 到 P40_1_2_3

代码：

```
set_spi_mode(SPI0, Spi_P40_1_2_3, Spi_End);
```

说明：

1.最后需要加入 Spi_End 作为结束标志

示例 2：发送一串 SPI 数据，并在接收的时候反转 P00 电平

场景：需要使用 SPI 发送数据的情况

代码：

```
char tx_tmp[7] = {0xab, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
```

```
...
void main(void){
    ...
    set_spi_mode(SPI0, Spi_End);
    spi_printf(SPI0, Buff_Mode, tx_tmp, 7);//使用 Buff 模式的示例
    While(1){
        if(get_spi_state(SPI0)){//操作完成
            P00 = ~P00;
        }
    }
}
```

说明：

1.使用 spi_printf 后会自动启动 dma 发送，发送完成后 get_spi_state(SPI0)返回 1

示例 3：读写 TLE5012 磁编码器，单线 SPI 收发的情况(SSC 接口)

场景：需要使用单线 SPI 收发，TLE5012 接在 SPI0 接口上

代码：

```
char tle5012[10] = {0x80,0x23,0xff,0xff,0xff,0xff,0xff,0xff,0xff, 0xff};
//发送读取角度、角速度、累计转数命令
unsigned int angle = 0, speed = 0, ppm = 0;
...
void main(void){
    ...
    // 设置磁编码器 SPI 部分
    set_io_mode(od_mode, Pin15, Pin_End);
    // 设置 MOSI 为开漏输出，方便单线读写，写 1 可释放总线
    set_io_mode(pp_mode, Pin14, Pin17, Pin_End);
    // 设置 SS,CLK 为推挽输出
    set_io_mode(high_speed, Pin14, Pin17, Pin_End);
    // 设置 SS,CLK 为高速输出
    set_io_mode(big_current, Pin14, Pin17, Pin_End);
    // 设置 SS,CLK 为大电流输出
    set_io_mode(en_pur, Pin14, Pin15, Pin16, Pin_End);
    // 打开 MISO、MOSI 的上拉电阻
    P14 = 0;// 拉低 CS 引脚
    set_spi_mode(SPI0, Low_Rising, Spi_End);
    spi_printf(SPI0, Buff_Mode, tle5012, 10);//使用 Buff 模式的示例
    While(1){
        if(get_spi_state(SPI0)){//操作完成，从缓冲区读取数据
            P14 = 0;// 读取完成，拉高 CS 引脚
            angle = *((unsigned int *)&_spi0_rx_buff[2]) & 0x7fff;
            speed = ((int)(*((unsigned int *)&_spi0_rx_buff[4])<<1))>>1;
            ppm = *((unsigned int *)&_spi0_rx_buff[6]) & 0x01ff;
        }
```

```
    }
}
```

说明：

1.使用 spi_printf 后会自动启动 dma 发送，发送完成后 get_spi_state(SPI0)返回 1

2.单线 SPI，只需要将 MOSI 和 MISO 连接到一起，然后 MOSI 设置为开漏模式

3.发送数据中，发送的数据为 0xff 即可自动放弃控制，交给 MISO 读取

4.TLE5012 是 CPHA=1，CPOL=0，所以这里更改了一下 CPOL(Low_Rising 参数)

5.Low_Rising 的意思是，默认低电平，第一个边沿是上升沿的意思

详细解释

SPI 外设说明

其中 SPI0 为独立的 SPI 外设，SPI1 和 SPI2 分别为串口 1 和串口 2 的同步 SPI 模式实现的，和标准 SPI 完全一致，但是使用的时候会占用串口 1 和串口 2，所以不要同时使用 SPI1 和串口 1，不要同时使用 SPI2 和串口 2。但同时使用串口 1 和 SPI2 是可以的。

PWM 部分

主要功能

可以自由设置 8 个 PWM 外设的输出周期和占空比。

其中，PWM1~4 是一组，PWM5~8 是一组，同组之间的频率必须相同，占空比可以不同。

主要函数

```
void set_pwm_mode(pwm_name pwm, ..., Pwm_End);
//可以设置 PWM 外设的周期占空比等参数，带有默认值和乱序输入支持
void set_pwm_duty(pwm_name pwm, float duty);
//可以设置 PWM 外设的占空比，想要改变周期请重新使用 set_pwm_mode，但是因为会重新计算周期的具体数值，所以使用 set_pwm_mode 时会消耗一定的时间
```

参数配置表

参数说明	可选参数	默认值	备注
PWM 外设编号	PWMx	-	范围 1~8
PWM 工作模式	Pwm_Out_Mode	Pwm_Out_Mode	输出模式
	Pwm_In_Mode		输入模式 （暂未支持）
PWM 通道使能	En_Out_P	En_Out_P	单独 P 通道输出
	En_Out_N		单独 N 通道输出
	En_Out_PN		PN 通道互补输出
	Dis_Out		关闭输出
设定 PWM 输出周期	“xkhz” “xhz”	“1khz”	支持两种单位，需全小写
设定 PWM 输出占空比	“x%”	50%	范围 0~100，支持浮点输入
设定互补输出死区	“xclk”	10clk	范围 0~1000clk
引脚切换	详情请见 set_pwm.h 文件		
PWM1~4 属于 PWMA 组，PWM5~8 属于 PWMB 组，每个 PWM 组只能设定同样的输出频率			

使用举例

示例 1：同时设置三路 PWM 为同样周期，并切换引脚

场景：用于驱动三相无刷电机的初始化情况

代码：

```
set_pwm_mode(Pwm1, Pwm1_P20_21, "20khz", "50%", Pwm_End);
    set_pwm_mode(Pwm2, Pwm2_P22_23, "50%", Pwm_End);
    set_pwm_mode(Pwm3, Pwm3_P24_25, "50%", Pwm_End);
```

说明：

1.最后需要加入 Pwm_End 表示参数结束

2.第一句设置过 PWMA 组周期后，后面可以不用再设置了

3.PWMA 组是 PWM1~PWM4，PWMB 组是 PWM5~PWM8

4.占空比支持小数点输入，输入”12.3%”这样也是可以的

5.切换引脚的时候注意要跟当前的 PWM 外设号对应

示例 2：输出带有互补的 PWM 波形

场景：用于驱动 H 桥电路，来控制有刷电机

代码：

```
set_pwm_mode(Pwm1, "1khz", "0%", En_Out_PN, Pwm_End);
```

说明：

1.En_Out_PN 为同时打开 P 通道和 N 通道(互补通道)

2.初始占空比设置为 0，以防止电机转动

3.PWM1 默认不设置的引脚为 Pwm1_P10_11

4.建议初始化后 PWM 外设再将 IO 模式从默认的高阻输入切换到推挽模式(推荐)

示例 3：同时输出 4 路 PWM 可驱动四路舵机

场景：多个舵机需要控制，需要都在同一个 PWM 组内，这里使用 PWMB 组

代码：

```
set_pwm_mode(Pwm5, "50hz", "2.5%", Pwm_End);
    set_pwm_mode(Pwm6, "2.5%", Pwm_End);
    set_pwm_mode(Pwm7, "2.5%", Pwm_End);
    set_pwm_mode(Pwm8, "2.5%", Pwm_End);
```

说明：

1.通用舵机驱动为 50hz 周期，高电平 0.5ms~2.5ms 控制对应角度

2.换算到百分比就是 2.5%~12.5%可以用来控制舵机的角度范围

3.只需要第一个设置过整组的 PWM 周期，后面就可以不用设置了

示例 4：修改指定 PWM 外设的占空比

场景：初始化完成后，需要继续控制占空比的情况

代码：

```
set_pwm_duty(Pwm1, 10.0f);
```

说明：

1.第一个参数是 PWM 外设号

2.后面是一个 float 类型的参数，用于指定 0~100%占空比

3.这行代码的意思是设置 PWM1 通道的占空比为 10%

详细解释

设置周期与占空比

可以设置 Pwm1~Pwm8 的周期和占空比，周期支持 khz 和 hz 单位，占空比支持 0%~100%，占空比支持小数点

例如占空比为 25.7%这种也是允许的。这里的占空比是初始化的占空比，后续如果还需要改变占空比

请使用 set_pwm_duty 函数，否则本函数重新计算并解析各个参数会较为耗费时间。

注意，Pwm1~4 为 PWMA 组，Pwm5~8 为 PWMB 组，设置其中一路 Pwm 通道的周期后，该组的其他通道设定时可以不用再重复设置周期。

如果设置了多次周期，按最后一次的设置的周期为准进行生效。

设置死区注意事项

支持死区的设置为 0~1000clk，clk 为 Pwm 的输入时钟，例如当前 Pwm 输入时钟 40Mhz，那么设置 100clk 的死区

死区的时间就是 1/40Mhz*100=2.5us，需要注意的是，死区设置在超过 127clk 后便不是非常精准的时钟数了。

因为原死区寄存器是 8 位的，在超过 127 的部分做了阶段性缩放，所以后面只能近似的进行设置死区时间。

下面进行一个 pwm 通道的设置举例：(周期的默认值为 1khz，占空比默认值为 50%， 死区默认值为 10clk)

```
set_pwm_mode(Pwm1, Pwm_Out_Mode, Pwm1_P10_11, En_Out_P, "1khz", "50%", "10clk", Pwm_End);
```

上面的程序代表的是 Pwm1 输出 1khz 的 50%占空比，死区为 10clk。并且只有 P 通道输出，N 通道不输出，切换到到 P10_P11 输出

当然，上面的这段设置程序也可以使用全默认值来设置：`set_pwm_mode(Pwm1, Pwm_End);`

可以根据自己想要设置的部分来进行设置，其他部分可以保留默认值

设置占空比注意事项

设置占空比，第一个参数是 Pwm1~Pwm8，第二个参数是占空比，支持小数点，范围 0%~100%

例如设置 24.5%的占空比，可以这么写：`set_pwm_duty(Pwm1, 24.5);`

EEPROM 部分

主要功能

用于存储一些掉电不想丢失的变量和核心数据

通过变量绑定机制实现快速的数据存储和拉取操作，内置 ADD8 校验和均衡磨损算法，无需关心底层实现。

主要函数

```
void set_eeprom_mode(const char *mode, void *value_addr, unsigned int len);
//用于绑定变量到 EEPROM，以实现类似掉电不变化的变量的效果。
void set_eeprom_sync(Push);
//用于将之前已经绑定的变量写入到 EEPROM 中
void set_eeprom_sync(Pull);
//用于将 EEPROM 中的数据再拉取到已经绑定的变量中
```

使用举例

示例 1：存储单片机开机次数

场景：用于指示当前单片机一共存在多少次开机上电复位

代码：

```
int start_cnt = 0;//开机次数存储变量
void main(void)
{
    EAXFR = 1; // 允许访问扩展寄存器
    WTST = 0;
    CKCON = 0;
    set_uart_mode(Uart1, Uart_End);
    EA = 1;//打开总中断
    set_eeprom_mode(Hex_Mode, &start_cnt, sizeof(start_cnt));
    //绑定变量到 EEPROM
    set_eeprom_sync(Pull);//拉取变量的值到本地变量
    start_cnt++;//开机次数加 1
    set_eeprom_sync(Push);//将本地变量的值更新到 EEPROM
    while(1)
    {
        if(get_uart_state(Uart1))//串口接收到任意数据
        {
            uart_printf(Uart1, "start_cnt:%d\r\n", start_cnt);
            //输出开机次数
        }
    }
}
```

说明：

1.进行 set_eeprom_sync()前应该先绑定变量

2.如果是初次上电，EERPROM 为空，第一次的 Pull 会失败，此时会导致 Pull 停止执行

3.每次 Push 都会自动执行均衡磨损，直到写满才会擦除

详细解释

EEPROM 基础操作函数和快捷使用函数的区别

EEPROM 基础操作函数是最基础的功能，可以使用 EEPROM 的所有操作，其中包括以下类型

EEPROM 操作函数对照表

操作名称	功能描述	等效函数原型
Read_Byte	读取 1 个字节	char set_eeprom_base(Read_Byte, unsigned long addr)
Read_Buff	读取多个字节	void set_eeprom_base(Read_Buff, unsigned long addr, char* buf, int len)
Write_Byte	写入 1 个字节	void set_eeprom_base(Write_Byte, unsigned long addr, char value)
Write_Buff	写入多个字节	void set_eeprom_base(Write_Buff, unsigned long addr, char* buf, int len)
Erase_Sector	擦除 1 个扇区	void set_eeprom_base(Erase_Sector, unsigned long addr)
Erase_Sectors	擦除多个扇区	void set_eeprom_base(Erase_Sectors, unsigned long addr, int len)

而 EEPROM 快捷使用函数是使用变量绑定机制的函数，会自动执行均衡磨损功能，内部还是调用的 EEPROM 基础操作函数

为何要使用自动均衡磨损？

首先需要区分真正的 EEPROM 和单片机的 Flash 模拟 EEPROM 的区别，真正的 EEPROM 支持单字节写，而 Flash 只支持整扇区擦除为 FF 后写 0。

正是因为 Flash 模拟出来的 EEPROM 没法单独擦除，所以才需要自动均衡磨损（就是每次需要写就直接在后面空白的地方继续写，直到全写满再擦除），使用了均衡磨损以后，EEPROM 的读写寿命可以提升数十倍，所以十分推荐这种 Flash 模拟的 EEPROM 来使用。