AI8051U 库函数使用说明

更新日期：2025 年 2 月 8 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));

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

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。

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。			

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
```

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。

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。

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

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。	

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。
```

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))

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。

详细解释

外部中断参数及触发类型表

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

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。
```

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);

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。

详细解释

定时器的设置是通过这几个参数实现的

定时器模式说明表

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	-	

而 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。

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。

主要函数

```
void set_uart_mode(uart_name uart, ...);
//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, ...);
//三模态打印函数，拥有普通 printf 功能、单字节 hex 发送功能、缓冲区方式发送功能。
```

详细解释

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。

举个例子：set_uart_mode(Uart1, "9600bps", Uart1_P36_7, Uart_End);

这个的意思是设置串口 1 为 9600 波特率，8 位数据位，1 位停止位，并切换引脚为 P36 和 P37 上，超时中断为 64byte

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。

再举个例子：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。

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。

参数配置表

参数类型	举例	格式	默认值
引脚切换宏定义	Uart1_P30_1	Uartx_Pxx_x	uart1 是 P30 和 P31 uart2 是 P12 和 P13 uart3 是 P00 和 P01 uart4 是 P02 和 P03
奇偶校验选择	Odd_9b	Mode_xb	Base_8b
使用定时器	Use_Timerx	Use_Timerx	Use_Timer2
设置波特率	“9600bps”	“xbps”	“115200bps”
设置超时字节数	“32byte”	“xbyte”	“64byte”

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 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

本 printf 自带长度校验和串口忙标志，超过长度会不打印，请到 set_uart.h 中改变 Uart_Tmp_Max

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。