UART 单线键盘接口驱动库

技术说明书

索引

综述

BC6xxx 系列键盘接口芯片和 BC759x 系列 LED 显示驱动+键盘接口芯片均提供统一的 UART 单线键盘接口，本驱动库可适用于所有使用该接口协议的芯片。使用本驱动库，用户仅需使用极少量(少于 10 行)的代码，即可实现包括按键按下/释放单独检测、组合键、长按键等功能在内的全功能键盘接口。

本驱动库使用标准 C 语言编写，可适用于所有使用 C 语言的目标环境。同时也是轻量型的驱动库，以 Cortex-M3 目标环境为例，本库编译后仅占用约 400 个字节的程序空间和 32 个字节的 RAM.

本驱动库与《BC727x 数码管及键盘接口芯片驱动库》在用户接口上高度一致，换用不同芯片时，用户程序仅需做微小的改动。

UART 单线键盘接口，每次传送一个 1 字节代表一个键盘事件，使用值 0-0x7F 代表按键键值，最高位作为按键释放标志，即 0-0x7F 表示按键按下，0x80-0xFF 表示按键释放，因此理论上最大的键盘数量为 128 键，实际芯片按键数量最多的是 BC7591 支持 96 键。本驱动库利用未使用的键值空间，让用户可以把组合键和长按键定义为自定义的键值，这样在用户程序中，处理组合键、长按键就变得和处理普通按键一样简单，仅需要根据不同键值进行程序分支，将所有复杂的转换和判断封装在了驱动库内。

因为组合键和长按键利用的是原始按键键值未利用的键值空间，因此有一定的限制，以 BC7591 96 键键盘为例，总共可供组合键和长按键利用的键值空间为 16 个。不过键值如果只要未被使用不会产生冲突，都可以定义为组合键和长按键使用，甚至可以不必是连续的键值空间。比如物理上 BC7591 使用了第 10-96 键，则第 0-9 的键值依然可以供用户自定义使用，这样就总共有 26 个键值供组合键和长按键使用。

驱动库提供了按键事件的缓存功能，因此用户程序可以在空闲时再查询键盘的状态，无需中断原本操作执行键盘响应，编程上大为简化。如果不使用组合键和长按键，驱动库的使用简化为仅需 3 步:

1. 第一步从 UART 中断中呼叫 update_key_status()函数。
2. 在主循环中呼叫 is_key_changed()函数查询是否有新的按键。
3. 呼叫 get_key_value()得到键值，执行相应按键响应操作。

当需要使用组合键或长按键时，仅需在上面基础上增加一步定义操作，告知驱动库组合键由哪些键组成、用什么键值代表该组合键即可，其余部分完全一致。

此外，驱动库支持动态改变设置，有特殊需求的用户，可以在运行时改变键盘检测模式、长按键时间、甚至组合键定义、长按键定义，是否使用回调函数(中断)相应键盘等。

函数参考手册

设置函数

set_detect_mode() 设置键盘检测模式

函数声明样式:

```
void set_detect_mode(unsigned char Mode);
```

此函数控制函数库的键盘检测模式。输入参数 Mode 为 0 时，本驱动库只报告按键的按下(导通)事件，而忽略按键的释放；Mode 为 1 时，驱动库将同时检测按键的按下和释放，一个键按下和释放的过程中，会两次令 is_key_changed() 查询的结果不为 0. 这里的按键，也包括用户自定义的组合键。

驱动库默认的工作状态为 Mode=0, 不检测按键释放。

set_longpress_count() 设置长按键计数值

函数声明样式:

```
void set_longpress_count(unsigned int CountLimit);
```

驱动库的长按键检测的时间，通过对 long_press_tick() 函数被呼叫的次数进行记次来实现。对设置了检测长按键的按键，如果在按键保持该状态的时间内，long_press_tick()被呼叫的次数超过了这里设置的 CountLimit 值，则驱动库就会报告一个长按键事件。

长按键计数值的默认值为 2000.

set_callback() 设置回调函数

函数声明样式:

```
void set_callback(void (*pCallbackFunc)(unsigned char));
```

此函数的输入参数为一具有 unsigned char 输入参数类型，返回类型为 void 的函数指针。当设置了此回调函数后，每当有新按键产生，都会自动呼叫此函数，而 is_key_changed() 查询，将不再能查询到新按键事件。回调函数被呼叫时，将以新按键的键值作为参数，用户可以在回调函数内完成按键的处理。传递的键值，也包括用户自定义的组合键和长按键的键值。

回调函数为用户提供了除“ is_key_changed() 查询 --> get_key_value() 获取键值”方式外的另外一种按键处理方式。回调函数方式在每次 update_key_status() 被呼叫时自动检查是否有新的按键事件，如果有，则会自动转去呼叫回调函数，对按键事件的处理可以做到没有延迟的及时处理。但如果 update_key_status() 函数的呼叫是发生在串口中断内，则回调函数也就成了串口中断服务程序的一部分，会造成串口中断所占用时间的延长。回调函数一般使用在用户程序因为所设计的工作方式无法及时查询键盘状态的情况下，比如用户程序为完全中断驱动方式等。

驱动库默认不呼叫回调函数，如果设置了回调函数后再次设置回调函数为 NULL，也会停止呼叫回调函数，恢复默认工作方式。

def_combined_key() 定义组合键

函数声明样式:

```
void def_combined_key(const unsigned char** pCBKeyList, unsigned char* pCBKeyMap, unsigned char CBKeyCount);
```

如果需要使用组合键，必须通过此函数将组合键的定义告知驱动库。本驱动库对组合键的工作方式，是由用户定义特定的键盘组合，并赋予一个特殊的键值，然后驱动库会监视键盘上这个特定的组合，当组合出现时，就会像对待普通按键一样，报告一个新按键事件。

组合键的定义，通过 3 个(种)数组完成:

组合键组成定义数组

定义每一个组合键由哪些键组成。数组的数据类型为 unsigned char，内容为以下格式: {count, defined_value, key1, key2, ... keyn} 其中，count 为该组合键所含的键数量，由 2 个键组成，则为 2, 由 3 个键组成，则为 3. 一个组合键，最多由 7 个按键组成。defined_value 是用户自定义的赋给该组合键的特殊键值，当组合键发生时，驱动库将报告一个新按键事件，而键值即为此 defined_value. 为避免和单独按键的键值冲突，此自定义键值应避开键盘芯片已经使用的键值，一般可在 97-126 的范围内选择，127(0x7F)号键值因为 0xFF 号键值被用做检测无按键操作的特殊用途，127 号的按键释放键值会与之冲突，不建议使用。key1, key2 ... keyn, 为组成组合键的各个键的原始键值。一个实际定义的例子:

```
const unsigned char cb1[] = {2, 125, 3, 19};
const unsigned char cb2[] = {3, 124, 4, 7, 10};
```

上面定义标明，名为 cb1 的组合键由 2 个键组成，分别为 3 号键和 19 号键，赋予的特殊键值为 125；名为 cb2 的组合键由 3 个键组成，分别为 4 号键，7 号键和 10 号键，赋予的特殊键值为 124.

此数组并不直接使用于 def_combined_key() 函数的呼叫，但是是下面组合键列表数组的成员，因此是必须的。

组合键列表数组

数组的类型为 unsigned char* , 其成员为组合键组合定义数组的列表。总共有多少个组合键，该数组就有多少个成员。以上面的组合键组成定义为例，有 cb1 和 cb2 两个组合键，组合键列表数组的定义为:

```
const unsigned char* cb_list[] = {cb1, cb2};
```

组合键映射数组

此数组的类型为 unsigned char，为提供给驱动库使用，供记录组合键各键状态用的存储空间。每个组合键，会使用其中的一个字节，因此该数组的大小，必须大于等于使用的组合键的数量。继续上面的例子，定义了两个组合键的系统，使用的组合键映射数组的定义为:

```
unsigned char key_map[2];
```

注意前两个数组的定义都有 const 关键字，因为前两个数组都是固定的数据，可以放在 ROM 中，而这个数组的定义则不使用 const 关键字，该数组必须存在于 RAM 内。

映射数组的内容，在呼叫 def_combined_key() 函数后被清零，对应所有按键释放(不导通)的状态。如果有成员按键在此时已经处于导通状态，就会形成映射数组不能正确反映键盘状态的情况。在这种情况下，即便组合键中其它按键随后也变为按下(导通)状态，也不会触发组合键事件，只有当这个不被正确反映的按键释放后，驱动库才能正确判断组合键的状态。

具有上面三个数组的定义后，就具备了呼叫本函数的全部参数条件，仍以上面的数据为例，呼叫本函数的方式为:

```
def_combined_key( cb_list, key_map, 2);
```

输入参数:

`pCBKeyList` ： 指向组合键列表数组的指针

`pCBKeyMap` ： 指向组合键映射数组的指针

`CBKeyCount` ： 组合键的数量

本驱动库可以支持在运行时改变组合键的定义，用户可以预先准备多套组合键定义，在运行时根据需要更换，不过如果在呼叫定义函数时组合键的某成员键不是在被释放的状态，则需要该按键释放后，驱动库才能正确识别组合键。

def_longpress_key() 定义长按键

函数声明样式:

```
void def_longpress_key(const unsigned char** pLPKeyList, unsigned char LPKeyCount);
```

定义长按键和定义组合键非常类似，不过长按键检测不需要用户提供 RAM 做键盘状态缓存，所以用户仅需要设置长按键定义和长按键列表 2 个数组就可以。函数的输入参数为两个，分别是:

`pLPKeyList`：指向长按键列表数组的指针

`LPKeyCount`：长按键的数量

传递长按键定义所需的 2 个数组:

长按键定义数组

定义长按键为哪个键，以及代表该键长按的用户自定义键值。数组的类型为 unsigned char，长度固定为 2 个字节，格式为 {key_value, defined_value}. 其中，key_value 为待检测长按键的按键的原始键值，这个键值，也可以是用户定义的组合键的键值，如果设置成组合键的键值，就可以完成对组合键的长按键检测。defined_value 为用户定义的代表该键长按的键值。自定义键值的选择和组合键自定义键值一样，应该选择使用不会与其它键冲突的键值。

每一个需检测长按的键，对应一个长按键定义数组，如果系统需要 2 个长按键，则需要定义 2 个长按键定义数组。如下例:

```
const unsigned char lp1[] = {4, 122};
const unsigned char lp2[] = {124, 121};
```

在上面的定义中，定义了两个长按键，第一个是 4 号键，是键盘上的独立物理按键，自定义键值 122,即该键长按时，驱动库会报告一个键值为 122 的新按键。第二个键待检测的是一个用户自定义的组合键，键值为 124, 长按键的自定义键值为 121. 其含义是如果用户自定义的键值为 124 的组合键长按了一定时间，驱动库会报告一个键值为 121 的新按键。如果沿用上面的例子，就是当物理 4 号，7 号和 10 号按键同时按下保持一定时间，就会产生一个键值为 121 的新按键事件。

长按键列表数组

数组类型为 unsigned char*，其内容为长按键定义数组的指针。数组元素的个数，就是系统中长按键的个数。使用上面的例子，则产生的列表数组为:

```
const unsigned char* lp_list[] = {lp1, lp2};
```

具有了长按键的定义数组后，就可以呼叫本函数通知驱动库所需要检测的长按键:

```
def_longpress_key(lp_list, 2);
```

上面函数的意义为：使用长按键列表 lp_list 中的长按键，总数量为 2 个。

定义长按键时，有一个特殊键值，专门用于代表无按键操作。如果将原始键值定义为 255(0xFF)，则代表对“无按键”的检测。如果最后一次按键释放后经过了设定的长按键时间内没有任何键盘动作，则驱动库就会报告一个键盘事件，键值为此处设定的自定义键值。检测“无按键”的定义数组例子:

const unsigned char no_key[] = {255, 255};

const unsigned char* lp_list[] = {lp1, lp2 ... no_key};

上面例子中，对“无按键”检测的自定义键值也设为了 255, 开启了长按键检测后，如果最后一次按键被释放后经过设定的长按键时间期间没有任何键盘操作，驱动库就会报告一个键值是 255 的键盘事件。

对“无按键”的检测，并不需要使能按键释放检测模式。

本驱动库设计为可以支持在运行时改变长按键的定义，用户可以预先准备多套长按键列表，在运行时改变长按键的设置。

输入函数

update_key_status() 更新键盘状态

函数声明样式:

```
void update_key_status(unsigned char RxData);
```

用来通知驱动库键盘的最新状态，该函数具有 1 个输入参数，类型为 unsigned char，为键盘芯片通过 UART 口传来的数据。

一般这个函数放在用户 UART 中断服务程序中，串口接收到数据后，直接呼叫此函数。不过这不是必须的方式，根据设计需要，用户可以从任何地方呼叫此函数。

这个函数被呼叫后，如果满足了新按键的条件，驱动库就会报告一个新键盘事件，is_key_changed() 就会返回非 0 的结果，或者如果设置成了使用回调函数的方式，在本函数返回前，回调函数就会被呼叫。

long_press_tick() 长按键计数

函数声明样式:

void long_press_tick(void);

长按键的检测，依赖于此函数。此函数应被定期呼叫。每次这个函数被呼叫，驱动库内部的计数器就加一，而每次 update_key_status() 函数被呼叫，该计数器则被清零。如果该计数器达到了设定的上限(通过 set_longpress_count()函数设置)，就会检查此前最后一个按键是否为待检测的长按键之一，如果是，就会报告一个长按键事件。

如果呼叫此函数的时间间隔固定，则长按键的时间，就是呼叫时间间隔×设定的计数上限。

此函数一般可以从定时器中断中呼叫，不过驱动库设计为可以从任何地方呼叫此函数。如果不使用定时器中断，也可以从程序主循环中呼叫。如果用户需要临时停止对长按键的检测，只需要停止呼叫此函数。

查询函数

is_key_changed() 查询是否有新按键

函数声明样式:

```
unsigned char is_key_changed(void);
```

用户程序通过此函数获得键盘的状态信息。程序返回值为 0 时，表示没有新的按键变化，返回非 0 时表示有新按键。新按键是指在当前工作模式下可被检测的按键变化，例如如果设置为不检测按键释放，则任何按键释放的事件不会影响此查询的结果。新按键同时包括组合键和长按键事件。

对于组合键，当组合成员中的每一个键按下时，都会如同单独的按键一样报告一个单键的新按键事件，但组合键中的最后一个键按下时，因为已经满足了组合键的条件，故只会报告组合键的新按键事件，而不会再单独产生该个体按键的按键事件。

当有两个键同时按下时，两个按键事件会连续产生，但如果在第一个按键事件产生后程序未能及时用get_key_value()获取键值，该事件将被第二个键盘事件替代。如果第一个按键是重要按键需要捕捉，建议将其和第二个键设置为组合键。

get_key_value() 获取键值

函数声明样式:

```
unsigned char get_key_value(void);
```

用户通过此函数获得键盘事件的键值。此函数可以在任何时候呼叫。此函数被呼叫后，如果此前有尚未获取的新按键，即 is_key_changed()函数返回非 0 结果，呼叫后 is_key_changed()则会恢复为返回 0.

本驱动库提供了键值锁存的功能，只要在下一个新按键到来之前，都可以通过本函数读取到最近一次的键值，因此减低了对用户程序实时性的要求。当设置为不检测按键释放时，两次连续按键的时间间隔通常最小为几百毫秒，用户程序可以有充分的时间处理当前的任务，待空闲时再对键盘做出反应。如果设置为按键按下和释放同时检测，则按键按下和释放都会产生键盘事件，这个时间间隔可能会缩短为几十毫秒。

如果当前键值未及时读取，当新的键盘事件发生时，新的键值会覆盖当前的键值。

如果使用回调函数处理按键，因为键值已经作为参数传递给回调函数，则无需再呼叫此函数。

使用方法示例

最简应用

此应用仅检测按键按下，不使用组合键和长按键，请参考以下伪代码(高亮部分为键盘相关代码):

```
#include “key_scan/key_scan.h”

int main(void)
{
    while(1)      // 主循环
    {
          do_something();     // 程序主任务
          …
          if (is_key_changed())      // 查询是否有新按键
          {
                Key_Value = get_key_value();      // 获取键值
                switch (Key_Value)
                {
                case 3:       // 3 号键所对应操作
                      ...
                      break;
                case 24:      // 24 号键所对应操作
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void UART_ISR(void)     // UART 中断处理程序
{
    if (RI)       // 如果是接收中断
    {
          update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
    }
    ...           // 其它串口操作
}
```

检测按键释放事件

请参考以下伪代码(高亮部分为较最简应用增加部分):

```
#include “key_scan/key_scan.h”

int main(void)
{
    set_detect_mode(1);         // 使能检测按键释放模式
    while(1)      // 主循环
    {
          do_something();     // 程序主任务
          …
          if (is_key_changed())      // 查询是否有新按键
          {
                Key_Value = get_key_value();      // 获取键值
                switch (Key_Value)
                {
                case 3:       // 3 号键所对应操作
                      ...
                      break;
                case 0x83:    // 0x83 是 3 号键释放对应的键值
                      ...
                      break;
                case 24:      // 24 号键所对应操作
                      ...
                      break;
```

```
                default:
                        break;
                }
        }
    }
}

void UART_ISR(void)     // UART 中断处理程序
{
    if (RI)       // 如果是接收中断
    {
        update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
    }
    ...           // 其它串口操作
}
```

长按键使用 - 使用定时器中断

请参考以下伪代码(高亮部分为较最简应用增加部分):

```
#include “key_scan/key_scan.h”

const unsigned char lp1[] = {5, 120};  // 长按键 1 为 5 号键，定义键值为 120
const unsigned char lp2[] = {20, 121}; // 长按键 2 为 20 号键，定义键值为 121
const unsigned char* LPDef[] = {lp1, lp2}; // 传递的长按键定义包括 lp1, lp2 两个

int main(void)
{
    set_longpress_count(60);          // 设置长按键时间为 3 秒(50ms*60)
    def_longpress_key(LPDef, 2);      // 定义 2 个长按键
    while(1)      // 主循环
    {
        do_something();     // 程序主任务
        …
        if (is_key_changed())      // 查询是否有新按键
        {
            Key_Value = get_key_value();      // 获取键值
            switch (Key_Value)
            {
            case 3:       // 3 号键所对应操作
                ...
                break;
            case 24:      // 24 号键所对应操作
                ...
                break;
            case 120:     // 5 号键长按所对应操作
                ...
                break;
            case 121:     // 20 号键长按所对应操作
                ...
                break;
            default:
                break;
            }
        }
    }
}

void UART_ISR(void)     // UART 中断处理程序
{
    if (RI)       // 如果是接收中断
    {
        update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
    }
    ...           // 其它串口操作
}

void TIMER_ISR(void)    // 定时器中断处理程序，每 50ms 中断一次
{
    long_press_tick();
}
```

长按键使用 - 不使用定时器中断

（高亮部分为与使用定时器方式区别）

```
#include “key_scan/key_scan.h”

const unsigned char lp1[] = {5, 120};  // 长按键 1 为 5 号键，定义键值为 120
const unsigned char lp2[] = {20, 121}; // 长按键 2 为 20 号键，定义键值为 121
const unsigned char* LPDef[] = {lp1, lp2}; // 传递的长按键定义包括 lp1, lp2 两个

int main(void)
{
    set_longpress_count(1500);          // 设置长按键时间为 3 秒(2ms*1500)
    def_longpress_key(LPDef, 2);        // 定义 2 个长按键
    while(1)        // 主循环
    {
        do_something();     // 程序主任务（假设耗时 2ms）
        …
        if (is_key_changed())       // 查询是否有新按键
        {
            Key_Value = get_key_value();      // 获取键值
            switch (Key_Value)
            {
            case 3:       // 3 号键所对应操作
                ...
                break;
            case 24:      // 24 号键所对应操作
                ...
                break;
            case 120:     // 5 号键长按所对应操作
                ...
                break;
            case 121:     // 20 号键长按所对应操作
                ...
                break;
            default:
                break;
            }
        }
        long_press_tick();
    }
}

void UART_ISR(void)     // UART 中断处理程序
{
    if (RI)       // 如果是接收中断
    {
        update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
    }
    ...           // 其它串口操作
}
```

组合键使用

请参见下面伪程序(高亮部分为较最简程序增加部分):

```
#include “key_scan/key_scan.h”

const unsigned char cb1[] = {2, 122, 0, 7};  // 组合键 1 由 2 个键组成(0 号键和 7 号键)，\
                                                定义键值为 122
const unsigned char cb2[] = {3, 123, 11, 15, 20}; // 组合键 2 由 3 个键组成(11,15,20 号)，\
                                                     定义键值为 123
const unsigned char* CBDef[] = {cb1, cb2}; // 传递的组合键定义包括 cb1, cb2 两个
unsigned char CBMap[2];    // 组合键检测所需要的映射用数组，数组大小须大于等于组合键的个数

int main(void)
{
    def_combined_key(CBDef, CBMap, 2);      // 传递 2 个组合键的定义
    while(1)      // 主循环
    {
        do_something();     // 程序主任务
```

```
                …
                if (is_key_changed())       // 查询是否有新按键
                {
                        Key_Value = get_key_value();      // 获取键值
                        switch (Key_Value)
                        {
                        case 3:         // 3 号键所对应操作
                                ...
                                break;
                        case 24:        // 24 号键所对应操作
                                ...
                                break;
                        case 122:       // 组合键 1 处理
                                ...
                                break;
                        case 123:       // 组合键 2 处理
                                ...
                                break;
                        default:
                                break;
                        }
                }
        }
}

void UART_ISR(void) // UART 中断处理程序
{
        if (RI)         // 如果是接收中断
        {
                update_key_status(RxData);          // 以所接收数据为参数呼叫 update_key_status 函数
        }
        ...             // 其它串口操作
}
```

组合键的长按键

(高亮部分为组合键的长按键相关部分)

```
#include “key_scan/key_scan.h”

const unsigned char lp1[] = {5, 120};  // 长按键 1 为 5 号键，定义键值为 120
const unsigned char lp2[] = {20, 121}; // 长按键 2 为 20 号键，定义键值为 121
const unsigned char lp3[] = {122, 125}; // 长按键 3 为组合键 1，定义键值为 125
const unsigned char* LPDef[] = {lp1, lp2, lp3}; // 传递的长按键定义包括 lp1, lp2, lp3 三个
const unsigned char cb1[] = {2, 122, 0, 7};  // 组合键 1 由 2 个键组成(0 号键和 7 号键)，\
                                                定义键值为 122
const unsigned char cb2[] = {3, 123, 11, 15, 20}; // 组合键 2 由 3 个键组成(11,15,20 号)，\
                                                     定义键值为 123
const unsigned char* CBDef[] = {cb1, cb2}; // 传递的组合键定义包括 cb1, cb2 两个
unsigned char CBMap[2]; // 组合键检测所需要的映射用数组，数组大小须大于等于组合键的个数

int main(void)
{
    set_longpress_count(60);          // 设置长按键时间为 3 秒(50ms*60)
    def_longpress_key(LPDef, 3);      // 定义 3 个长按键
    def_combined_key(CBDef, CBMap, 2);      // 传递 2 个组合键的定义
    while(1)      // 主循环
    {
            do_something();     // 程序主任务
            …
            if (is_key_changed())       // 查询是否有新按键
            {
                    Key_Value = get_key_value();      // 获取键值
                    switch (Key_Value)
                    {
                    case 3:         // 3 号键所对应操作
                            ...
                            break;
                    case 24:        // 24 号键所对应操作
                            ...
                            break;
                    case 120:       // 5 号键长按所对应操作
                            ...
                            break;
```

```
                    case 121:     // 20 号键长按所对应操作
                           ...
                           break;
                    case 122:     // 组合键 1 处理
                           ...
                           break;
                    case 123:     // 组合键 2 处理
                           ...
                           break;
                    case 125:     // 组合键 1 长按所对应操作
                           ...
                           break;
                    default:
                           break;
                    }
             }
      }
   }

   void UART_ISR(void)      // UART 中断处理程序
   {
      if (RI)        // 如果是接收中断
      {
             update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
      }
      ...            // 其它串口操作
   }

   void TIMER_ISR(void)     // 定时器中断处理程序，每 50ms 中断一次
   {
      long_press_tick();
   }
```

回调函数使用(中断驱动型程序)

请参考以下伪代码(高亮部分为键盘相关代码):

```
#include "key_scan/key_scan.h"

void key_action(unsigned char Key)
{
       switch(Key)
       {
       case 2:       // 2 号键处理
              ...
              break;
       case 3:       // 3 号键处理
              ...
              break;
       default:
              break;
       }
}

int main(void)
{
       set_callback(key_action);          // 定义回调函数
       eni();        // 使能中断
       while (1)
       {
              sleep();      // CPU 始终处于睡眠状态，所有操作由中断完成
       }
}

void UART_ISR(void) // UART 中断处理程序
{
       if (RI)       // 如果是接收中断
       {
              update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
       }
       ...           // 其它串口操作
}
```

检测长时间无按键

检测长时间无按键，并不需要使能按键释放检测，请参考以下伪代码(高亮部分为“无按键”检测相关代码):

```
#include “key_scan/key_scan.h”

const unsigned char lp1[] = {5, 120};  // 长按键 1 为 5 号键，定义键值为 120
const unsigned char lp2[] = {20, 121}; // 长按键 2 为 20 号键，定义键值为 121
const unsigned char no_key[] = {255, 255}; // "无按键“检测专用长按键定义
const unsigned char* LPDef[] = {lp1, lp2,  no_key}; //  传递的长按键定义包括 lp1, lp2, no_key 三个

int main(void)
{
    set_longpress_count(60);          // 设置长按键时间为 3 秒(50ms*60)
    def_longpress_key(LPDef, 3);      // 定义 3 个长按键
    while(1)      // 主循环
    {
          do_something();     // 程序主任务
          …
          if (is_key_changed())      // 查询是否有新按键
          {
                Key_Value = get_key_value();      // 获取键值
                switch (Key_Value)
                {
                case 3:       // 3 号键所对应操作
                      ...
                      break;
                case 24:      // 24 号键所对应操作
                      ...
                      break;
                case 120:     // 5 号键长按所对应操作
                      ...
                      break;
                case 121:     // 20 号键长按所对应操作
                      ...
                      break;
                case 255:     // “无按键”事件
                      no_key_time++;       // 每发生一次进入睡眠计数器加一
                      if (no_key_time >= 10)     // 如果达到 10 倍的长按键时间没有键盘操作
                      {
                            no_key_time = 0;
                            sleep();      // 进入睡眠模式
                      }
                      break;
                default:
                      break;
                }
                if (Key_Value != 255)       // 如果有任何其它按键事件
                {
                      no_key_time = 0;     // 进入睡眠计数器回零
                }
          }
    }
}

void UART_ISR(void)      // UART 中断处理程序
{
    if (RI)       // 如果是接收中断
    {
          update_key_status(RxData);        // 以所接收数据为参数呼叫 update_key_status 函数
    }
    ...           // 其它串口操作
}

void TIMER_ISR(void)     // 定时器中断处理程序，每 50ms 中断一次
{
    long_press_tick();
}
```