BC727x

数码管及键盘接口芯片

驱动库

技术说明书

索引

综述

BC727x 系列 LED 显示和键盘接口芯片提供统一的 SPI 接口和指令，本驱动库可适用于所有该系列芯片。使用本驱动库，用户不仅可以通过短短 1 条语句完成常用的显示十进制数字、显示 16 进制数字、闪烁控制等显示功能，同时仅需使用极少量(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。

驱动库可使用于使用硬件 SPI 接口和软件模拟 SPI 接口两种方式。

如果用户程序周期性地更新显示，间隔在 100ms 以内，使用本驱动库可以免去对芯片上 KEY 信号的监视和处理，节省 MCU 一根口线以及相关处理代码。

本驱动库与面向 BC6xxx 和 BC759x 系列芯片的《UART 单线键盘接口驱动库》在用户接口上高度一致，在换用不同系列芯片时，用户程序仅需更改涉及硬件变更的部分。

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，本库显示和键盘接口合并编译后仅占用约 900 个字节的程序空间和 24 个字节的 RAM(非中断模式下)。

BC727x 使用 SPI 接口，本驱动库可以支持 8 位和 16 位两种 SPI 接口，同时可以配置为不使用中断和使用中断两种模式。

不使用中断接口简单，用户代码量少，产生的驱动库代码也小；使用 SPI 中断，库内自建 FIFO 缓冲区，通讯效率高，所占用的处理器时间短，适合 CPU 任务繁重的情况下使用。

驱动库代码分为显示和键盘两部分，显示部分除了提供基础的 send_cmd()函数可供向 BC727x 芯片发送任何指令外，还提供了几个上层函数，可完成最常用的显示功能：显示十进制数字、显示 16 进制数字、设置字符闪烁。

键盘接口部分，将从 SPI 接收到的 16 位键盘映射做运算，将键盘的变化直接转换为易于用户处理的键值。不仅普通的按键直接给出键值，而且用户可以自定义组合按键和长按键，包括检测组合键的长按键，也都转换为特定的键值，用户可以用统一的方式处理不同的按键事件。

键值的转换规则很简单，16 位键盘映射中，bit0 对应键值为 0, bit1 对应为 1,直至 bit15 键值为 15(0x0F). 用户可以自定义最多 16 个组合键，每个组合键可以由任意多个键组成，用户定义的第一个组合键，键值为 0x10，第二个为 0x11，依次类推，直至 0x1F. 而用户定义的长按键，是指驱动库会监视的有长按行为的按键，长按键可以是单键或组合键。长按键的键值是按键原本键值 +0x20, 如 3 号键的长按键键值是 0x23，而第二个组合键(0x11)的长按键键值将是 0x31.

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，键盘的使用简化为仅需 3 步：

1. 第一步收到 SPI 读回的键盘映射后呼叫 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. 如有新按键呼叫 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、需要监视长按的是哪些键，按键处理部分与单键处理保持一致。

此外，驱动库键盘部分支持动态改变设置，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。

为了适用于不同的芯片，本驱动库需要用户提供几个函数供驱动库呼叫，主要是和 SPI 硬件接口相关的函数，最基本的函数仅需要一个:

1. SPI 口写入函数。

用户必须提供一个供数据写入 SPI 口的函数，如果使用软件模拟 SPI 口，则为 SPI 的发送函数。

如果使用了 CS，则需要提供给驱动库控制 CS 电平的方法:

2. CS 口线置高/置低函数

因为 BC727x 的 SPI 口在指令和数据之间必须保持 CS 低电平，在 SPI 8 位模式时，由用户自行控制 CS 会比较繁琐，CS 操作不能放在 SPI 写入函数中，需要在每次调用函数库操作前都进行 CS 操作，更简便的方式是给驱动库提供 CS 操作的函数，由驱动库来控制 CS 电平。当使用中断时，驱动库提供了 FIFO 缓冲区，用户呼叫显示函数后会立刻返回，而此时数据才开始发送，因此必须由驱动库来完成 CS 的控制。用户须将 CS 的操作封装成函数提供给驱动库。

当系统中使用了多片 BC727x 芯片需要使用 CS 信号来分别选中不同芯片时，可以设置多套 CS 口线操作函数，在对不同芯片操作前，切换驱动库做控制的 CS 操作函数，达到让驱动库控制不同芯片的目的。

如果希望使用中断方式操作 SPI 外设，则用户还需要提供额外的几个函数给驱动库：SPI 发送中断使能/禁止函数

3. 使用 SPI 中断时，驱动库需要控制中断的使能和禁止，用户需要将具体的操作封装成函数提供给驱动库。

本驱动库为可配置的驱动库，配置通过配置文件 bc727x_config.h 来完成。可供选择的配置包括:

- 数码管排列方向（可适用于不同布局的电路设计）
- 是否使用键盘
- SPI 口数据宽度 8 位或 16 位
- 是否使用 SPI 中断模式
- 中断模式下 FIFO 的大小

驱动库的配置

驱动库的配置通过 bc727x_config.h 文件完成，可配置内容包括 5 项，如果某一项的内容缺失，会使用默认值。

数码管排列方向

```
#define LOW_DIG_NUM_ON_RIGHT 1
```

设定数码管是从右向左排列还是从左向右排列。如果编号较低的数码管位是位于右侧，则此值定义为 1。此设置影响数字显示函数的显示方向。比如用 disp_dec()函数从 DIG4 开始显示十进制数，个位显示于 DIG4，此值为 1 时，十位将显示在 DIG5，而 LOW_DIG_NUM_ON_RIGHT 为 0 时，十位将显示在 DIG3. 此设置使驱动库可以适应不同的电路板设计。

是否使用键盘

```
#define USE_KEY_SCAN 1
```

如果不使用驱动库的键盘功能，将 USE_KEY_SCAN 定义为 0, 编译的结果即不会包含键盘相关内容。

SPI 口工作模式

```
#define SPI_MODE_INTERRUPT 0
```

此项默认值为 0. 如果 SPI 接口使用中断模式，将此值定义为 1，编译结果将包含 SPI 中断操作相关函数。

SPI 口数据宽度

```
#define SPI_MODE_16BIT 0
```

默认值为 0, 此时将产生适用于 8 位 SPI 接口的代码，如果定义为 1,则会产生对应 16 位 SPI 口的代码。

SPI 口 FIFO 大小

```
#define SPI_FIFO_SIZE (4)
```

默认值为 4, 此值必须为 2 的指数倍，如 2, 4, 8, 16, 32 等等。注意实际的缓冲区的字节数受 SPI 数据宽度影响，当数据宽度为 16 位时，缓冲区占用的字节数会加倍。

函数参考手册

显示函数

clear() 清除显示和闪烁

函数声明样式:

```
void clear(void);
```

清除所有的显示，并同时清除所有的闪烁属性。

send_cmd() 向BC727x 芯片发送指令

函数声明样式:

```
void send_cmd(uint8_t Cmd, uint8_t Data);
```

此函数为基础的 BC727x 操作，可以用来向 BC727x 芯片发送任何指令。其它的显示函数都通过调用此函数实现。第一个参数 Cmd 为指令，也即 BC727x 的寄存器地址；第二个参数 Data 为待写入寄存器的数据。

display_dec() 以10 进制显示数值

函数声明样式:

```
void display_dec(uint32_t Val, uint8_t Pos, uint8_t Width);
```

本指令以第 Pos 位数码管为最低位(个位)按 10 进制显示数值 Val，Val 的范围是 0～4294967295。Pos 的取值范围是 0-15。如果数值的位数超出了可显示的位数，超出部分不会显示，函数也不会报告错误。Width 的低 7 位为显示的宽度，最高位 bit7 控制当数字实际位数小于 Width 设定时是否显示前导'0'。如果 Width 小于实际数字的宽度，超出部分被忽略，如果 Width 大于实际数字宽度，当 Width 最高位为 0 时，超出部分将显示空白(小数点亦将被清除)；当 Width 最高位为 1 时，超出部分将显示 0, 这对于显示如时间的分秒等要求固定显示位数的应用比较有用。

驱动库的配置选项中 LOW_DIG_NUM_ON_RIGHT 项控制此函数数值显示的方向。当该项值为 1 时，从个位开始显示使用的数码管位递增，如 Pos 设置为 3, 则个位显示在 DIG3，而十位将显示在 DIG4. 当 LOW_DIG_NUM_ON_RIGHT 设置为 0, 上面例子里十位将显示在 DIG2.

display_hex() 以16 进制显示数值

函数声明样式:

```
void display_hex(uint16_t Val, uint8_t Pos, uint8_t Width);
```

功能与上面函数类似，不过数值将以 16 进制显示。此函数增加了一个输入参数 Digits, 该参数决定显示的位数。Val 的取值范围为 0-0xFFFF, 不过 Width 的数值不限于 0-4, 如果设置的显示位数 Digits 超过

了实际数值的长度，显示仍会按设置的显示位数显示，超出的位将会显示 0. 例如输入 Val 的值为 0x2A, Width 设置为 6, 则显示将为 00002A. 如果 Width 的值小于待显示数值的位数，则只会显示 Width 范围内的位，如上例中如果 Digits 设置为 1, 则仅会显示'A'。

欲显示超过 16 位的 16 进制数值，可以将其拆分然后分两次调用此函数。

此函数也受 LOW_DIG_NUM_ON_RIGHT 设置的影响。

display_float() 显示浮点数

函数声明样式:

```
void display_float(float Val, uint8_t Pos, uint8_t Width, uint8_t Frac);
```

此函数直接接受 float 数据类型的值 Val，并显示出来，小数点后的最低位自动四舍五入。小数点后面的位数，由 Frac 决定，Frac 取值范围为 0-7；整体的显示区域位置和宽度，由 Pos 和 Width 决定，其含义同于 display_dec()函数中的相应参数。Width 必须不小于 Frac。**本函数并不会显示小数点和负号**，有需要时用户需用额外指令完成小数点和负号的显示。这样当小数点位置固定时，用户可以使用固定电路驱动小数点常亮而将芯片输出的小数点驱动用作其它指示灯。

下面通过举例说明控制参数的用法(为了直观，例子中显示结果包含小数点):

例 1：如 Val=3.14，Width=5, Frac=4, 则显示结果为 3.1400

例 2：如 Val=3.14159, Width=7, Frac=4, 则显示结果为 ␣␣3.1416

例 3：如 Val=0.00567, Width=0x88, Frac=3, 则显示结果为 00000.006

例 4：如 Val=476.0056, Width=4, Frac=3, 则显示结果为 6.006 (高位因超出显示宽度未显示)

本函数内部实际是将待显示浮点数转换为 32 位无符号整数然后调用 display_dec()函数实现的，因此必须注意防止 32 位整数溢出的问题。如有浮点数 Val=500000.3，如果使用 Frac=4 为参数显示，则函数内部需先将数据转换为无符号整数 5000003000，但 32 位无符号整数所能表示最大值为 4,294,967,295，因此将产生溢出而无法得到正确结果。

digit_blink() 数码管按位闪烁

函数声明样式:

```
void digit_blink(uint8_t Digit, uint8_t OnOff);
```

该函数控制数码管显示位 Digit 的闪烁，如果 OnOff 值为 1,则该位闪烁，如果为 0,则停止闪烁。对目标位的闪烁控制，不会影响其它位的闪烁属性。

此函数每次只控制一个显示位，如果想一次控制多个位的闪烁属性，可以直接操作芯片内的位闪烁控制寄存器。

设置函数

set_write_func() 设置(注册)写SPI 口函数

函数声明样式:

```
void set_write_func(void (*pSpiWriteFunc)(uint8_t));
```
(SPI 口 8 位模式时)

```
void set_write_func(void (*pSpiWriteFunc)(uint16_t));
```
(SPI 口 16 位模式时)

用户必须通过此函数告知驱动库调用哪个函数来写入 SPI 口，才能使驱动库正常运作。不同的器件 SPI 口的硬件设计会有不同，数据宽度也会有 8 位和 16 位两种。当硬件上有 8 位和 16 位两种模式可选择时，优先选用 16 位模式，因为这样函数的调用次数会减少一倍，效率更高。

用户提供的 SPI 写入函数必须具有 1 个输入参数，根据 SPI 的数据宽度，为 8 位或 16 位数据；返回参数必须是 void.

写入 SPI 口通常是一个简单的向 SPI 外设的寄存器写入数值的操作，用户需要确保该寄存器处在可写的状态。数据写入寄存器后，不必等待数据发送完成，函数即可返回。

这个 SPI 口写入函数，也包括软件模拟 SPI 接口的方式。在软件模拟 SPI 方式下，函数需要完成数据的移位、输出后，才能返回。

set_cs_low_func() , set_cs_high_func() 设置(注册)CS 操作函数

函数声明样式:

```
void set_cs_low_func(void (*pCSSetLFunc)(void));
void set_cs_high_func(void (*pCSSetHFunc)(void));
```

电路中使用 CS 信号时，可由驱动库来控制 CS 信号的电平。

使用 FIFO 时因为无法区分缓冲区内数据的归属，因此使用中断模式下，驱动库无法控制多路 CS 信号，从而只能用于 SPI 总线上只有 1 片 BC727x 芯片的情况,

CS 线置高和 CS 线置低必须是两个独立的函数，必须具有 void 输入参数和 void 返回参数。

如果不使用 CS 信号，或使用类似 BC7278 这样无 CS 信号的芯片时，程序可跳过设置 CS 操作函数的步骤，这样函数库会自动跳过呼叫这两个函数，或者将操做函数定义为空函数。

这两个函数可以在任何时候呼叫，也可以重复呼叫。当系统中使用了多片 BC727x 芯片，需要控制不同的 CS 信号线来选中不同的芯片时，可以准备多套 CS 操作函数，分别对应不同芯片的操作，在需要切换操作的芯片时，重新设置驱动库的 CS 操作函数，即可达到多片联用的目的。需要注意在使用中断模式时，因为驱动库内置有 SPI 口 FIFO 缓冲区，切换 CS 前，需要确保缓冲区内的数据均已经发送完成。可

以通过监视当前所使用的 CS 信号线电平来判断，驱动库会在开始发送前将 CS 置低，直到缓冲区内数据全部发送完成，才将其恢复为高电平，通过读取当前 CS 线的电平，即可知道缓冲区是否已经发送完成。

set_eni_func(), set_disi_func() 设置(注册)SPI 发送中断使能/禁止函数

函数声明样式:

```
void set_eni_func(void (*pTxIntEnFunc)(void));
void set_disi_func(void (*pTxIntDisFunc)(void));
```

这两个函数也是只用于使用中断模式的情况下。 驱动库进行中断操作时，需要临时禁止和使能中断，用户必须提供这两个函数用来完成这两个操作。如果所使用的芯片 SPI 的中断区分为发送和接收两个中断，这里所提供的应该是发送中断的使能/禁止函数。

**** 以下设置函数只有使用键盘功能时才需要 ****

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. 这里指的按键，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 40.

set_callback() 设置回调函数

函数声明样式:

```
void set_callback(void (*pCallbackFunc)(uint8_t));
```

此函数的输入参数为一具有 uint8_t 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 SPI 中断内，则回调函数也就成了中断服务程序的一部分，会造成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 uint16_t CBKeyList[], uint8_t
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。本驱动库处理组合键的工作方式，是由用户定义特定的键盘组合，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 1 个数组完成:

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 uint16_t. 实际即为组合键的键盘映射，16 位对应 16 个键，组合键的成员为 1, 非成员为 0. 如设置 1 号和 3 号键为组合键，则数据为 0x000A(0x000A=0b00000000 00001010)合键可以由任意多个键组成。

为避免和单独按键的键值冲突，组合键的键值从 0x10 开始，依次递加，如果定义了 4 个组合键，则键值将分别是 0x10, 0x11, 0x12, 0x13. 最多可以定义 16 个组合键。

一个实际定义的例子:

```
const uint16_t cb_keys[] = {0x000A, 0x0009, 0x8003};
```

上面定义表明，定义了 3 个组合键，前两个由 2 个键组成，分别为 1 号键和 3 号键，0 号键和 3 号键；第三个组合键由 3 个键组成，为第 0 号、1 号和 15 号。

使用上面定义的例子，函数在呼叫时用如下格式:

```
def_combined_key(cb_keys, 3);
```

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。

def_longpress_key() 定义长按键

函数声明样式:

```
void def_longpress_key(const uint8_t LPKeyList[], uint8_t
LPKeyCount);
```

定义长按键和定义组合键非常类似，不过长按键定义数组内提供的是需要检测长按键的按键键值，而不是键盘映射。长按键定义数组的数据类型是 uint8_t, 长按键的数量不受限制，可以有任意多个。

待检测长按键的键值，可以是单个按键的键值，也可以是组合键的键值。有一个特殊的键值 0xFF, 用来检测无按键的时间。 如果设置的长按键定义里有 0xFF, 且键盘持续了长按键的时间保持所有键都是释放的状态，则驱动库就会产生一个按键事件，键值为 0xFF。假如设置的长按键时间为 3 秒，需要检测 30 秒无按键操作进入节能状态，只需要设置长按键定义包含 0xFF，然后如果程序连续收到了 10 个 0xFF 的按键，则说明已经连续 30 秒没有任何键盘操作。

除了 0xFF 以外，其它定义的长按键的键值，是其原本键值+0x20.

定义实例:

```
const uint8_t lp_keys[] = { 0x01, 0x03, 0x10, 0xff};
```

上面定义了 4 个长按键，分别是 1 号键，3 号键，第一个组合键，以及无按键检测。函数使用时，语法如下:

```
def_longpress_key(lp_keys, 4);
```

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

tx_ready() SPI 发送缓冲区准备好

函数声明样式:

```
void tx_ready(void);
```

只有在 SPI 中断模式下，才需要使用此函数。此函数的作用在于通知驱动库，可以向 SPI 写入下一个数据了。一般从 SPI 的中断中呼叫这个函数。

MCU 的 SPI 口，在数据发送完成可以接受下一个数据时，会产生一个中断，这个函数一般在 SPI 中断里呼叫。

spi_check_end() 检查SPI 是否发送结束

函数声明样式:

```
void spi_check_end(void);
```

这个函数同样只有在 SPI 中断模式下才需要，这个函数用来让驱动库检查是否应该恢复 CS 的高电平，所以如果没有使用 CS 片选功能，也无需呼叫这个函数。其次，如果使用了键盘功能，这个函数已经包含在了下面 update_key_status() 中，因此如果使用了键盘功能用户也无需使用此函数。

如果使用，这个函数应该在 SPI 的数据发送完成，因为 SPI 的数据发送和接收是同步的，因此也是数据接收完成的时候呼叫，通常是在数据接收完成的中断里。总结起来这个函数只有在以下几个条件均满足时才需要由用户呼叫:

- 使用中断模式
- 使用了 CS 片选
- 没有使用键盘功能

**** 以下函数只有使用键盘功能时才需要 ****

update_key_status() 更新键盘状态

函数声明样式:

```
void update_key_status(uint16_t KeymapData);
```

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 uint16_t，为每次 SPI 通讯时收到的键盘映射。因此，这个函数一般是放在 SPI 接收的中断中，或者不使用中断时在 SPI 发送结束收到了新的键盘映射时呼叫。

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；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。

如果设置为 SPI 中断模式，此函数还包含了对 spi_check_end() 的呼叫。

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，而每次从 SPI 收到的键盘映射发生变化，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，呼叫的时间间隔也不一定固定。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。

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。

键值的分配，可以参考下表:

按键类型	按键按下(导通)的键值	按键释放的键值
单一按键	键盘映射中每个键的键值即其 bit 位值，bit0 对应键值 0, bit15 键值为 15.	键值+128(最高位置 1)
组合键	组合键的序号+16(0x10)	
长按键	被检测按键的键值+32(0x20)	不产生键盘事件
无按键	0xFF	

如果一个按键按下后，在释放之前又有新的按键按下，如果这两个键的组合被定义为了组合键，此时除了第一个按键会报告一个键值为单一按键的键盘事件，第二个键按下时，会报告组合键的按键事件,而不会报告第二个单独按键。如果这两个键没有定义为组合键，则会产生一个新的单一按键键盘事件。

如果两个键严格在同一时间按下，当定义了组合键，会正常报告组合键事件，如果是两个独立按键，则驱动库只会报告键号比较小的按键。这种情况下如果用户需要确保不错失第二个按键，必须读取键盘映射由用户自行判断。

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

get_key_map() 获取当前键盘映射

函数声明样式:

```
uint16_t get_key_map(void);
```

驱动库将键盘的状态变化转换为单一的键值，使得键盘的处理变得简单,但尽管可以设置组合键，有时用户会有特殊需要掌握整个键盘的状态，此时可以使用 `get_key_map()` 函数，此函数返回最后一次从 `SPI` 接收到的键盘映射值。

使用方法示例

以下涉及键盘的例子中，均假定用户程序会按照一定的频率更新显示内容，间隔在 100ms 内，这种情况下可以不检测 KEY 引脚的状态，因为键盘映射会不断更新给驱动库。如果没有一定频率的显示更新操作(如果显示内容没有变化，确实不需要刷新显示，因为 BC727x 芯片可保持显示内容，不刷新可以节约 CPU 时间)，则需要按照最后例子使用 KEY 引脚中断，保证及时的键盘响应。

最简应用

此应用检测按键按下，然后将键值按 16 进制显示在第 0，1 两位，不使用组合键和长按键。请参考以下伪代码(高亮部分为驱动库相关代码):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    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
                display_hex(Key_Value, 0, 2);     // 显示键值
                switch (Key_Value)
                {
                case 3:       // 3 号键所对应操作
                      ...
                      break;
                case 4:       // 4 号键所对应操作
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat);  // 更新键盘状态
}
```

使用 16 位 SPI 接口

与最简应用相同，但使用 16 位 SPI 口(高亮部分为与 8 位模式区别):

```
在 bc727x_config.h 中，包含：
#define SPI_MODE_16BIT 1


#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    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
                  display_hex(Key_Value, 0, 2);     // 显示键值
                  switch (Key_Value)
                  {
                  case 3:       // 3 号键所对应操作
                        ...
                        break;
                  case 4:       // 4 号键所对应操作
                        ...
                        break;
                  default:
                        break;
                  }
            }
    }
}

void spi_send(uint16_t Dat)    // SPI 发送程序
{
    uint16_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}
```

使用软件模拟 SPI

与上例相同，但不使用硬件 SPI 口，而使用软件模拟 SPI. （高亮部分为与使用硬件 SPI 区别部分)

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);         // 设置 SPI 发送函数
    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
                  display_hex(Key_Value, 0, 2);     // 显示键值
                  switch (Key_Value)
                  {
                  case 3:       // 3 号键所对应操作
                        ...
                        break;
                  case 4:       // 4 号键所对应操作
                        ...
                        break;
                  default:
                        break;
                  }
            }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t i, RxDat;
    delay_half_T();      // 延时半个时钟周期(>7.8us)
    for (i=0; i<8; i++)
    {
            if (Dat&0x80)        // 输出 Dat 最高位到 MOSI
            {
                  MOSI = 1;
            }
            else
            {
```

```
                MOSI = 0;
            }
            CLK = 0;
            delay_half_T();      // 延时半个时钟周期(>7.8us)
            CLK = 1;
            delay_half_T();
            Dat = Dat<<1;        // Dat 左移 1 位
            RxDat == RxDat<<1;   // 从 MISO 读入 1 位数据到 RxDat
            if (MISO)
            {
                RxDat = RxDat|0x01;
            }
    }
    update_key_status(RxDat); // 更新键盘状态
}
```

使用 CS 信号

在最简应用基础上增加对 CS 信号的控制(高亮部分为 CS 相关代码):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    set_cs_high_func(cs_set_1);
    set_cs_low_func(cs_set_0);
    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
            display_hex(Key_Value, 0, 2);     // 显示键值
            switch (Key_Value)
            {
            case 3:       // 3 号键所对应操作
                ...
                break;
            case 4:       // 4 号键所对应操作
                ...
                break;
            default:
                break;
            }
        }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}

void cs_set_1(void)
{
    CS = 1;
}

void cs_set_0(void)
{
    CS = 0;
}
```

控制多个 CS 信号(多片联用)

当驱动库同时控制多片 BC727x 芯片时(高亮部分为与基本 CS 控制相比较增加部分):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    uint16_t     i, j;
    set_write_func(spi_send);         // 设置 SPI 发送函数

    while(1)      // 主循环
    {
        use_cs1();                    // 控制第一片 BC727x
        display_dec(i++, 0, 4);       // 在第一片芯片第 0 位显示 i,宽度为 4 位
        use_cs2();                    // 切换为控制第二片 BC727x
        display_dec(j--, 0, 5);       // 在第二片芯片第 0 位显示 j，宽度为 5 位
        …
        if (is_key_changed())         // 查询是否有新按键
        {
            Key_Value = get_key_value();      // 获取键值
            display_hex(Key_Value, 7, 2);     // 显示键值(依然在第 2 片芯片，显示在第 7 位)
            switch (Key_Value)
            {
            case 3:       // 3 号键所对应操作
                ...
                break;
            case 4:       // 4 号键所对应操作
                ...
                break;
            default:
                break;
            }
        }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}

void use_cs1(void)       // 切换为控制 CS1
{
    set_cs_high_func(cs1_set_1);
    set_cs_low_func(cs1_set_0);
}

void use_cs2(void)
{
    set_cs_high_func(cs2_set_1);
    set_cs_low_func(cs2_set_0);
}

void cs1_set_1(void)
{
    CS1 = 1;
}

void cs1_set_0(void)
{
    CS1 = 0;
}

void cs2_set_1(void)
{
    CS2 = 1;
}

void cs2_set_0(void)
{
    CS2 = 0;
}
```

使用 SPI 中断方式并使用 CS 信号

在最简应用基础上改为使用 SPI 中断方式，同时使用 CS 信号(高亮部分为所变化的相关代码):

```
在 bc727x_config.h 中，包含：
#define SPI_MODE_INTERRUPT 1

#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(write_spi); // 设置 SPI 发送函数
    set_cs_low_func(cs_set_0);          // 设置 CS 控制函数
    set_cs_high_func(cs_set_1);
    set_eni_func(enable_spi_int);             // 设置中断控制函数
    set_disi_func(disable_spi_int);
    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
            CS = 0;
            display_hex(Key_Value, 0, 2);     // 显示键值
            CS = 1;
            switch (Key_Value)
            {
            case 3:       // 3 号键所对应操作
                ...
                break;
            case 4:       // 4 号键所对应操作
                ...
                break;
            default:
                break;
            }
        }
    }
}

void write_spi(uint8_t Dat)     // SPI 发送程序
{
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
}

void cs_set_0(void)
{
    CS = 0;       // 设置 CS 线为低
}

void cs_set_1(void)
{
    CS = 1;       // 设置 CS 线为高
}

void enable_spi_int(void)
{
    SPI_IE = 1;   // 使能 SPI 中断
}

void disable_spi_int(void)
{
    SPI_IE = 0;   // 禁止 SPI 中断
}

void SPI_ISR(void)       // SPI 中断服务函数，SPI 数据发送（同时也是接收）完成时触发
{
    update_key_status(SPI_BUF);       // 用接收到的数据呼叫 update_key_status 函数
    tx_ready();                       // 通知驱动库可发送下一个数据
```

```
}
```

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 “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    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
                display_hex(Key_Value, 0, 2);     // 显示键值
                switch (Key_Value)
                {
                case 0x03:    // 3 号键按下所对应操作
                      ...
                      break;
                case 0x83:    // 3 号键释放所对应操作
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}
```

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 “bc727x_lib/bc727x.h”

const uint8_t LPDef[] = {2, 5}; // 长按键定义包括 2 号键和 5 号键两个

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    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 4:       // 4 号键所对应操作
                      ...
```

```
                        break;
                case 0x22:    // 2 号键长按所对应操作
                        ...
                        break;
                case 0x25:    // 5 号键长按所对应操作
                        ...
                        break;
                default:
                        break;
                }
        }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}

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 “bc727x_lib/bc727x.h”

const uint8_t LPDef[] = {2, 5}; // 长按键定义包括 2 号键和 5 号键两个

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    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 4:       // 4 号键所对应操作
                    ...
                    break;
            case 0x22:    // 2 号键长按所对应操作
                    ...
                    break;
            case 0x25:    // 5 号键长按所对应操作
                    ...
                    break;
            default:
                    break;
            }
        }
        long_press_tick();
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
```

```
    while (SPI_BUSY);   // 等待数据发送完成
    RxDat = SPI_BUF;    // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}
```

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 “bc727x_lib/bc727x.h”

const uint16_t CBDef[] = {0x0003, 0x0c00}; // 定义 2 个组合键，分别是 0 号+1 号，10 号+11 号键

int main(void)
{
    set_write_func(spi_send);         // 设置 SPI 发送函数
    def_combined_key(CBDef, 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 0x10:    // 组合键 1 处理
                      ...
                      break;
                case 0x11:    // 组合键 2 处理
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // 数据写入 SPI 寄存器
    while (SPI_BUSY);   // 等待数据发送完成
    RxDat = SPI_BUF;    // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}
```

组合键的长按键

(高亮部分为组合键的长按键相关部分)

```
#include “bc727x_lib/bc727x.h”

const uint16_t CBDef[] = {0x0003, 0x0c00}; // 2 个组合键定义，分别是 0 号+1 号，10 号+11 号键
const uint8_t LPDef[] = {0x10};            // 长按键为第一个组合键(键值 0x10)


int main(void)
{
    set_write_func(spi_send);         // 设置 SPI 发送函数
    set_longpress_count(60);          // 设置长按键时间为 3 秒(50ms*60)
    def_longpress_key(LPDef, 1);      // 定义 1 个长按键，为第一个组合键
    def_combined_key(CBDef, 2);       // 定义 2 个组合键
    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 4:       // 4 号键所对应操作
                ...
                break;
            case 0x30:    // 第一个组合键长按所对应操作
                ...
                break;
            default:
                break;
            }
        }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // 数据写入 SPI 寄存器
    while (SPI_BUSY);   // 等待数据发送完成
    RxDat = SPI_BUF;    // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}
```

回调函数使用

请参考以下伪代码(高亮部分为相关代码):

```
#include “bc727x_lib/bc727x.h”

void key_action(unsigned char Key)      // 键盘处理函数
{
    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);          // 设置回调函数
    set_write_func(spi_send);          // 设置 SPI 发送函数
    while(1)      // 主循环
    {
        do_something();     // 程序主任务
        …
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // 数据写入 SPI 寄存器
    while (SPI_BUSY);   // 等待数据发送完成
    RxDat = SPI_BUF;    // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
```

```
}
```

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 “bc727x_lib/bc727x.h”

const uint8_t LPDef = {0x02, 0xff}; // 长按键定义为 2 号键和无按键

int main(void)
{
    set_write_func(spi_send);          // 设置 SPI 发送函数
    set_longpress_count(60);           // 设置长按键时间为 3 秒(50ms*60)
    def_longpress_key(LPDef, 2);       // 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 4:       // 4 号键所对应操作
                      ...
                      break;
                case 0xff:    // “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 != 0xff)      // 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 spi_send(uint8_t Dat)      // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}

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

显示更新(SPI 操作)不频繁时通过 KEY 信号达到及时键盘响应

前面的键盘应用例程均假定用户程序会定期更新显示内容，更新的间隔在 100ms 内。因为每次 SPI 口的操作都会同时更新键盘的状态，因此驱动库可以及时了解键盘的状态变化，不必检查 KEY 引脚。但如果程序设计上访问间隔比较长，单纯靠被动获得键盘映射会造成对键盘反应迟钝，也有可能丢键。这

时就需要靠检测芯片上的 KEY 信号来及时了解键盘的变化。通常 KEY 引脚会连接一个外部中断，当 KEY 的电平发生变化，即引起中断，而中断处理中，可发送伪指令 0xFF，强迫 SPI 口工作，从而更新键盘状态，报告按键事件。

因为外部中断中会使用驱动库函数使用 SPI 口发送伪指令，而如果中断发生在主程序中使用同一个 SPI 口的操作的过程中，会发生资源冲突，因此，如果使用 KEY 引脚触发外部中断，则应该在主程序中进行 BC727x 操作时，临时禁止外部中断；或者在外部中断的服务程序中调用发送伪指令函数之前，判断当前 SPI 是否正在使用中，如果正在使用中则跳过(发送伪指令的目的在于强迫 SPI 口工作以得到最新键盘映射，如果 SPI 口已经在工作中，则无需这个操作了)。

如果驱动库的 SPI 口操作是使用的中断模式，则来自外部中断的 bc727x_send_cmd()操作可能会扰乱 SPI FIFO 缓冲区的读写。好在中断模式已经要求用户提供了执行使能/禁止 SPI 中断操作的函数，仅需要在这两个函数中增加同时对外部中断的使能/禁止即可。

伪代码例程(高亮部分为相关代码):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);         // 设置 SPI 发送函数
    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
                disable_EXTI();                   // 临时禁止外部中断
                display_hex(Key_Value, 0, 2);     // 显示键值
                enable_EXTI();                    // 重新使能外部中断
                switch (Key_Value)
                {
                case 3:       // 3 号键所对应操作
                      ...
                      break;
                case 4:       // 4 号键所对应操作
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI 发送程序
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // 数据写入 SPI 寄存器
    while (SPI_BUSY);    // 等待数据发送完成
    RxDat = SPI_BUF;     // 获取 SPI 接收到的数据
    update_key_status(RxDat); // 更新键盘状态
}

void EXTI_ISR(void)      // 外部中断处理程序，当 KEY 引脚发生变化时产生此中断
{
    EXTIF = 0;           // 清除中断标志
    send_cmd(0xff, 0xff);      // 发送伪指令
}
```