# BC727x

# LED Display & Keyboard Interface Driver Library

# User Manual

# Index

# Introduction

The BC727x series LED display and keyboard interface chips provide a unified SPI interface and instruction set, and this driver library can be used for all chips in this series. By using this driver library, users can easily accomplish common display functions such as displaying decimal and hexadecimal numbers, and controlling blinking with just one line of code. Additionally, with less than 10 lines of code, users can implement a full-featured keyboard interface, including individual detection of key presses/releases, combination keys, and long-press keys.

This driver library can be used with both hardware SPI interfaces and software simulated SPI interfaces. If the user program updates the display periodically with intervals less than 100ms, this driver library can eliminate the need for monitoring and handling the ‘KEY’ signal on the chip, saving one MCU I/O and related processing code.

This driver library is highly consistent with the "UART Single-Line Keyboard Interface Driver Library" for the BC6xxx and BC759x series chips in terms of user interface. When switching to a different series of chips, the user program only needs to modify the parts involving hardware changes.

The driver library is written in standard C language and is compatible with all target environments that use C language. It is also a lightweight library. For example, in the Cortex-M3 target environment, the combined program space for the display and keyboard interfaces is only about 900 bytes and 24 bytes of RAM (in non-interrupt mode).

The BC727x uses an SPI interface, and this driver library can support both 8-bit and 16-bit SPI interfaces, and can be configured to use interrupt or non-interrupt modes. The non-interrupt interface is simple, with less user code and smaller generated driver library code. Using SPI interrupts, the library has a built-in FIFO buffer, which has high communication efficiency and occupies less processor time, making it suitable for situations with heavy CPU tasks.

The driver library code is divided into two parts: display and keyboard. The display section provides not only the basic "send_cmd()" function for sending any command to the BC727x chip but also several upper-level functions for the most commonly used display functions: displaying decimal and hexadecimal numbers, and setting character blinking. The keyboard interface section performs operations on the 16-bit keyboard mapping received from the SPI, directly converting changes in the keyboard into key values that are easy for users to process. It not only provides key values for ordinary keys, but also allows users to define combination keys and long-press keys, including detecting long-presses for combination keys, which are all converted to specific key values that users can handle in a unified manner.

The conversion rules for key values are simple. In the 16-bit keyboard mapping, bit0 corresponds to the key value 0, bit1 corresponds to 1, and so on, with bit15 corresponding to 15 (0x0F). Users can define up to 16 combination keys, each of which can consist of any number of keys. The first combination key defined by the user has a key value of 0x10, the second is 0x11, and so on, up to 0x1F. Long-press keys defined by the user are keys that the driver library monitors for long-press behavior, and can be single keys or combination keys. The key value for a long-press key

is the original key value plus 0x20. For example, the key value for the long-press key of key 3 is 0x23, and the key value for the long-press key of the second combination key (0x11) is 0x31.

The driver library provides a cache function for key events, allowing the user program to query the keyboard status during idle time without interrupting the current operation and simplifying programming significantly. If composite keys and long press keys are not used, the use of the keyboard is simplified to just three steps:

1. Call the `update_key_status()` function after receiving the keyboard mapping from the SPI readback.
2. Call the `is_key_changed()` function in the main loop to check if there are any new keys.
3. If there are new keys, call `get_key_value()` to obtain the key value and execute the corresponding key response operation.

When composite keys or long press keys are needed, simply add a step to define the operation by telling the driver library which keys are included in the combination keys and which keys need to be monitored for long presses. The key processing part is the same as that for single keys.

In addition, the keyboard section of the driver library supports dynamic configuration changes, allowing users with special needs to change the keyboard detection mode, long press key time, combination key definition, long press key definition, and whether to use callback functions (interrupts) to respond to the keyboard during runtime.

To be suitable for different chips, this driver library requires users to provide several functions for the driver library to call, mainly related to the SPI hardware interface. The most basic function only requires:

1. SPI write function. The user must provide a function to send data to the SPI interface, which becomes the SPI sending function if a software simulated SPI interface is used.

If CS is used, a method for controlling the CS level must be provided to the driver library:

2. CS line set high/low function. Since the SPI interface of BC727x must maintain a low level of CS between instructions and data, controlling CS by the user can be cumbersome in 8-bit mode. The CS operation cannot be placed in the SPI write function and must be performed before each call to the function library operation. It is more convenient to provide the driver library with a function for controlling CS operations to let the driver library control the CS level. When using interrupts, the driver library provides a FIFO buffer. After the user calls the display function, the function returns immediately, but the data is sent afterward. Therefore, the driver library must have the CS control. The user must encapsulate the CS operation into a function to provide it to the driver library.

If interrupt-driven operation of the SPI peripheral is desired, the user must provide several additional functions to the driver library:

3. SPI send interrupt enable/disable function. When using SPI interrupts, the driver library needs to control the enable and disable of the interrupt, and the user needs to encapsulate specific operations into a function to provide them to the driver library.

This driver library is configurable and can be configured using the configuration file bc727x_config.h. The available configuration options include:

- Display orientation (adapt different PCB layouts)
- Whether to use the keyboard
- SPI interface data width (8-bit or 16-bit)
- Whether to use the SPI interrupt mode
- FIFO size under interrupt mode

# Library Configuration

The configuration of the driver library is done through the bc727x_config.h file and includes 5 configurable options. If any of these options are missing, default values will be used.

## Display Orientation

```
#define LOW_DIG_NUM_ON_RIGHT 1
```

This option sets the direction of the numerical display, either from right to left or from left to right. If the lower numbered digit is on the right side, this value is defined as 1. This setting affects the display direction of the numerical display functions. For example, using the `disp_dec()` function to display a decimal number starting from DIG4, if this value is 1, the tens digit will be displayed on DIG5, while if it is 0, the tens digit will be displayed on DIG3. This setting allows the driver library to adapt to different circuit board designs.

## Use of Keyboard

```
#define USE_KEY_SCAN 1
```

If the keyboard functionality of the driver library is not required, define USE_KEY_SCAN as 0. The resulting compilation will not include any keyboard-related content.

## SPI Mode

```
#define SPI_MODE_INTERRUPT 0
```

The default value is 0. If the SPI interface uses interrupt mode, define this value as 1, and the resulting compilation will include SPI interrupt operation-related functions.

## SPI Data Width

```
#define SPI_MODE_16BIT 0
```

The default value is 0, which produces code suitable for 8-bit SPI interfaces. If defined as 1, code corresponding to a 16-bit SPI interface will be produced.

## SPI FIFO Size

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

The default value is 4, which must be a power of 2, such as 2, 4, 8, 16, 32, etc. Note that the actual number of bytes used by the buffer is affected by the SPI data width. When the data width is 16 bits, the number of bytes used by the buffer will double.

# Function Reference Manual

## Display Functions

### *clear() Clear Display and Blink*

Function Declaration:

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

Clears all displays and clears all blink attributes at the same time.

### *send_cmd() Send Command to BC727x*

Function Declaration:

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

This function is the basic BC727x operation and can be used to send any instruction to the BC727x chip. Other display functions are implemented by calling this function. The first parameter, Cmd, is the instruction, which is the BC727x register address; the second parameter, Data, is the data to be written to the register.

### *display_dec() Display a Decimal Number*

Function Declaration:

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

This function displays the decimal number `Val` starting from the `Pos` digit, where `Pos` is the lowest digit (the units digit) of the display. `Val` ranges from 0 to 4294967295. `Pos` ranges from 0 to 15. If the number of digits in `Val` exceeds the number of digits that can be displayed, the excess digits will not be displayed, and the function will not report an error. The low 7 bits of `Width` are the display width, and the highest bit, bit 7, controls whether leading zeros are displayed when the number of digits is less than `Width`. If `Width` is less than the actual width of the number, the excess digits are ignored. If `Width` is greater than the actual width of the number and bit 7 of Width is 0, the excess digits will be blank (the decimal point will also be cleared); if bit 7 of `Width` is 1, the excess digits will be displayed as 0. This is useful for applications that require a fixed number of displayed digits, such as displaying time in minutes and seconds. The `LOW_DIG_NUM_ON_RIGHT` option in the library configuration controls the direction in which this function displays numbers. When this option is set to 1, the display digits used for displaying the number increase from the units digit, so if `Pos` is set to 3, the units digit will be displayed on DIG3 and the tens digit will be displayed on DIG4. When `LOW_DIG_NUM_ON_RIGHT` is set to 0, the tens digit in the above example will be displayed on DIG2.

### *display_hex() Display a Hexadecimal Number*

Function Declaration:

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

This function is similar to the previous function, but the number is displayed in hexadecimal. `Val` ranges from 0 to 0xFFFF, but Width is not limited to 0-4. If `Width` is set to a value larger than the actual length of the number, the number will still be displayed according to the specified number of digits, and the excess digits will be displayed as 0. For example, if `Val` is 0x2A and `Width` is set to 6, the display will be 00002A. If Width is less than the number of digits to be displayed, only the digits within the range of `Width` will be displayed. For example, if `Width` is set to 1 in the above example, only 'A' will be displayed.

To display a hexadecimal number with more than 4 digits, split it and call this function twice.

This function is also affected by the LOW_DIG_NUM_ON_RIGHT setting.

### *digit_blink() Blink Individual Digits*

Function Declaration:

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

The function digit_blink() controls the blinking of the specified digit on the display. If OnOff is set to 1, the digit will blink; if it is set to 0, the blinking will stop. This function only affects the specified digit and will not affect the blinking property of other digits.

Note that this function can only control one digit at a time. If you want to control the blinking property of multiple digits at once, you can directly operate the digit blinking control register in the chip.

## Setup Functions

### *set_write_func() Setting (registering) The SPI Write Function*

Function Declarations:

`void set_write_func(void (*pSpiWriteFunc)(uint8_t));` (for 8-bit SPI mode)

`void set_write_func(void (*pSpiWriteFunc)(uint16_t));` (for 16-bit SPI mode)

Users must use this function to inform the library which function to call to write to the SPI interface in order to make the driver library work properly. Different hardware designs for the SPI interface on different devices will have different data widths, either 8 bits or 16 bits. When both 8-bit and 16-bit modes are available on the hardware, the 16-bit mode is preferred because it reduces the number of function calls by half and is more efficient.

The SPI write function provided by the user must have one input parameter, either an 8-bit or 16-bit data depending on the SPI data width, and must return void.

Writing to the SPI interface is usually a simple operation of writing a value to a register in the SPI peripheral, and the user needs to ensure that the register is in a writable state. After writing the data to the register, the function can return without waiting for the data to be sent.

This SPI write function also includes the software simulation mode of the SPI interface. In software simulation mode, the function needs to complete data shifting and output before it can return.

### *set_cs_low_func() , set_cs_high_func() Set (register) CS Operation Functions.*

Function Declarations:

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

When using the CS signal in the circuit, the driver library can control the level of the CS signal. When using the FIFO, it is not possible to distinguish the ownership of the data in the buffer, so the driver library cannot control multiple CS signals in interrupt mode and can only be used in the case where only one BC727x chip is present on the SPI bus.

Setting the CS line to low and high must be two independent functions that have void input and return parameters. If the CS signal is not used or a chip like BC7278 without a CS signal is used, the program can skip the step of setting the CS operation function. In this case, the function library will automatically skip calling these two functions, or the operation functions can be defined as empty functions.

These two functions can be called at any time and can be called repeatedly. When multiple BC727x chips are used in the system and different CS signal lines need to be controlled to select different chips, multiple sets of CS operation functions can be prepared, each corresponding to the operations of different chips. To switch to the operation of a different chip, the driver library's CS operation functions need to be reset. When using interrupt mode, because the driver library has a SPI port FIFO buffer, it is necessary to ensure that all data in the buffer has been sent before switching CS. The driver library will set the CS to low before starting transmission and restore it to a high level only after all data in the buffer has been sent. By reading the current level of the CS line, it can be determined whether the buffer has been sent completely.

### *set_eni_func(), set_disi_func() Setting (registering) SPI Interrupt Enable/Disable Functions*

Function Declarations:

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

These two functions are used only in interrupt mode. When the driver library performs interrupt operations, it needs to temporarily enable or disable interrupts. Users must provide these two functions to complete these operations. If the SPI of the chip being used distinguishes between

sending and receiving interrupts, the function provided here should be the one that enables/disables sending interrupts.

**** The following setting functions are only required when using the keyboard function. ****

### *set_detect_mode() Set Keyboard Detection Mode*

Function Declaration:

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

This function controls the keyboard detection mode of the library. When input parameter `Mode` is 0, the library only reports key press (closed) events and ignores key releases. When Mode is 1, the library will also detect key releases(open). During a key press and release process, the function `is_key_changed()` will return a non-zero result twice. Here, the term "key" also includes user-defined combination keys.

The default working mode of the library is `Mode=0`, which does not detect key releases. This function is only needed when keyboard function is used.

### *set_longpress_count() Set The Count Limit For Long Press Keys Detection.*

Function Declaration:

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

The library detects long press of keys by counting the number of times the `long_press_tick()` function is called. For a key that is set to detect long press, if the `long_press_tick()` function is called more than `CountLimit` times within the time period during which the key is held, the library will report a long press key event.

The default value for the long press count limit is 40.

### *set_callback() Set Callback Function*

Function Declaration:

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

This function sets a callback function to be automatically called whenever a new key event occurs, including user-defined combination keys and long press keys. The function pointer passed to this function must take a `uint8_t` input parameter and have a `void` return type. When the callback function is set, `is_key_changed()` queries will no longer be able to detect new key events. Whenever a new key event occurs, the callback function will be automatically called with the new key value as its parameter. The user can then perform key press event processing in the callback function.

The callback function provides an alternative way to handle key events, as opposed to the "`is_key_changed()` query -> `get_key_value()` get key value" approach. The callback function is automatically checked for new key events every time `update_key_status()` is

called, allowing for timely processing of key events without delay. However, if `update_key_status()` is called within an SPI interrupt, the callback function will become part of the interrupt service routine and will extend the interrupt's handling time.

By default, the driver library does not call the callback function. If a callback function is set and then set to NULL again, the driver library will stop calling the callback function and resume its default operation.

## *def_combined_key() Define Combination Keys*

Function Declaration:

```
void def_combined_key(const uint16_t CBKeyList[], uint8_t CBKeyCount);
```

If you need to use combination keys, you must inform the driver library of the definition of the combination keys through this function. The way this driver library handles combination keys is by allowing users to define specific keyboard combinations, and then the driver library will monitor this specific combination on the keyboard. When the combination appears, it will report a new key event just like a normal key.

The definition of the combination key is completed through an array:

### Key Combination Definition Array

Defining the composition of each combination key is necessary for using combination keys. The data type of the array is `uint16_t`, which corresponds to the keyboard mapping of the combination key, with each bit corresponding to a key. If keys 1 and 3 are set as a combination key, the data is 0x000A. A combination key can consist of any number of keys.

To avoid conflicts with the values of individual keys, the key values of combination keys start from 0x10 and be increased by each definition. If four combination keys are defined, the key values will be 0x10, 0x11, 0x12, and 0x13. Up to 16 combination keys can be defined.

As an example, the following code defines three combination keys: the first two consist of two keys, keys 1 and 3, and keys 0 and 3, respectively; the third consists of three keys, keys 0, 1, and 15:

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

To use the above example, the function should be called in the following format: `def_combined_key(cb_keys, 3);`

This library supports changing the definition of combination keys during runtime. Users can prepare multiple sets of combination key definitions and switch them on the fly.

## *def_longpress_key() Define Long-press Keys*

Function Declaration:

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

Defining long press keys is very similar to defining combined keys, except that the long press key definition array provides the key values that need to be checked for long press, rather than the keyboard mapping. The data type of the long press key definition array is `uint8_t`, and the number of long press keys is unlimited and can be any number.

The key value to be checked for long press can be the key value of a single key or the key value of a combined key. There is a special key value of `0xFF` used to detect no key press. If the long press key definition set includes `0xFF` and all keys are released for the duration of the long press time, the driver will generate a key event with a key value of `0xFF`. If the long press time is set to 3 seconds and a power-saving mode is to be entered after 30 seconds of no key press, just set the long press key definition to include `0xFF`. If the program receives 10 consecutive `0xFF` key presses, means there has been no keyboard operation for 30 seconds.

Except the 0xFF, the key value of other defined long press keys is its original key value + 0x20.

Here is an example of a long press key definition array:

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

This defines four long press keys: key 1, key 3, the first combined key, and a no-key-press detection. The syntax for using the function is as follows: `def_longpress_key(lp_keys, 4);`

Detecting "no key press" does not require enabling the key release detection mode.

This driver is designed to support changing the long press key definition at runtime. Users can prepare multiple sets of long press key lists and change the long press key settings at runtime.

# Input Functions

## *tx_ready() SPI Send Buffer Ready*

Function declaration:

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

This function is only required when using the SPI interrupt mode. Its purpose is to notify the driver library that the next data can be written to the SPI. Generally, this function is called from the SPI interrupt.

When the MCU's SPI port can receive the next data after the data transmission is completed, it generates an interrupt. This function is generally called within the SPI interrupt.

## *spi_check_end() Check SPI Sending Is Completed*

Function declaration:

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

This function is only required when using SPI interrupt mode. Its purpose is to notify the driver that the next data can be written to the SPI. It is usually called from the SPI interrupt. When the data

transfer is complete, the MCU's SPI generates an interrupt indicating that the next data can be transmitted. This function is typically called from the SPI interrupt. However, if CS chip selection is not used, there is no need to call this function. Furthermore, if the keyboard function is used, this function is included in the `update_key_status()` function. Therefore, the user does not need to use this function if the keyboard function is used. If used, this function should be called when data is received because SPI data transmission and reception are synchronous. Thus, it is usually called from the data reception interrupt. In summary, this function is only required if all of the following conditions are met:

- Interrupt mode is used
- CS chip selection is used
- The keyboard function is not used.

**** The following functions are only needed when using the keyboard function ****

### *update_key_status() Update Keyboard Status 更新键盘状态*

Function declaration:

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

This function is used to notify the driver library of the latest keyboard status. It has one input parameter, which is a uint16_t type and represents the keyboard mapping received during each SPI communication. Therefore, this function is usually placed in the SPI receive interrupt, or called when a new keyboard mapping is received after the SPI transmission ends when interrupt is not used.

After this function is called, if the conditions for a new key press are met, the driver library will report a new keyboard event, and `is_key_changed()` will return a non-zero result. If the callback function is set to be used, the callback function will be called before this function returns.

If set to SPI interrupt mode, this function also includes a call to `spi_check_end()`.

### *long_press_tick() Long-press Ticks*

Function declaration:

```
void long_press_tick(void);
```

This function is essential for detecting long press keys and should be called periodically. Each time this function is called, the internal counter in the driver library is incremented, and the counter is cleared whenever a new keyboard mapping is received from SPI. If the counter reaches the set limit (set by the `set_longpress_count()` function), the driver library checks whether the last pressed key is one of the long press keys to be detected. If so, a long press key event is reported.

If the time interval for calling this function is fixed, the duration of the long press key is the time interval multiplied by the set count limit.

This function can generally be called from a timer interrupt, but the function is designed to be called from anywhere and the calling interval does not necessarily need to be fixed. If timer interrupts are not used, the function can also be called from the main program loop. If the user needs to temporarily stop the detection of long press keys, simply stop calling this function.

# Query function

## *is_key_changed() Query for new keystrokes*

Function declaration:

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

This function is used by the user program to obtain the status information of the keyboard. If the return value of the function is 0, it means that there is no new key change. If it is non-zero, it means that there is a new key. New keys refer to key changes that can be detected under the current operating mode. For example, if the function is set not to detect key releases, any key release events will not affect the query result. New keys also include combination keys and long press key events.

For combination keys, when each key in the combination is pressed, a single new key event will be reported as if it were a single key. However, when the last key in the combination is pressed and the condition for the combination key is satisfied, only the new key event of the combination key will be reported, and the key event of the individual key will not be generated again.

## *get_key_value() Get Key Value*

Function declaration:

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

Users can use this function to obtain the key value of a keyboard event at any time. This function can be called at any time. After this function is called, if there is an unread new key event, i.e. `is_key_changed()` returns a non-zero result, `is_key_changed()` will return 0 again after the call.

This library provides key value caching function, so the most recent key value can be obtained through this function until the next new key arrives. This reduces the real-time requirements of user programs. When set to ‘not detect key release’, the time interval between two consecutive key presses is usually several hundred milliseconds, giving the user program sufficient time to handle the current task before responding to the keyboard when the task is finished. If set to ‘detect both key press and release’, both key press and release will generate keyboard events, and the time interval may be shortened to several tens of milliseconds. If the current key value is not read in a timely manner, when a new keyboard event occurs, the new key value will overwrite the unread key value.

Key value assignment can refer to the following table:

| Key type | Key value when key is pressed | Key value when key is released |
|---|---|---|
| Single key | The key value of each key in the keyboard mapping corresponds to the bit value of the key, with bit 0 corresponding to key value 0 and bit 15 corresponding to key value 15. | Key value + 128 (highest bit set to 1) |
| Combination key | Combination key number + 16 (0x10) | |
| Long press key | Key value of the detected key + 32 (0x20) | No event is generated |
| No key | 0xFF | |

If a key is pressed and then another key is pressed before the first one is released, if the combination of these two keys is defined as a combo key, only the first key will report a single key keyboard event, and when the second key is pressed, it will report a combo key event instead of the second individual key. If these two keys are not defined as a combination key, a new single key keyboard event will be generated.

If two keys are pressed at exactly the same time, if a combo key is defined, a combo key event will be reported normally. If they are two independent keys, the driver will only report the key with the smaller key number. In this case, if the user needs to ensure that the second key is not missed, they must read the keyboard mapping and judge it themselves.

If a callback function is used to handle the key, the key value has already been passed as a parameter to the callback function, and there is no need to call this function again.

### *get_key_map() Get Current Keyboard Mapping*

Function declaration:

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

The driver library converts the keyboard's state changes into a single key value, making keyboard processing simple. However, although it is possible to set combination keys, sometimes the user may have a special need to understand the entire keyboard state. In this case, the get_key_map() function can be used, which returns the last keyboard mapping value received from SPI.

# Usage Examples

In the following examples involving the keyboard, it is assumed that the user program updates the display content at a certain frequency, within 100ms intervals. In this case, it is not necessary to check the status of the KEY pin, as the keyboard mapping is continuously updated to the driver library. If there is no frequent display update operation (if the display content does not change, it is not necessary to refresh the display, as the BC727x chip can maintain the display content and save CPU time), then it will be necessary to use the KEY pin interrupt according to the last example to ensure timely keyboard response

## Simplest Application

This application detects key presses and displays the key value in hexadecimal on the first two digits, without using combination keys or long presses. Please refer to the following pseudocode (highlighted parts are related to the driver library)

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);          // Set SPI sending function
    while(1)      // Main loop
    {
          do_something();     // Main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();      // Get key value
                display_hex(Key_Value, 0, 2);     // Display key value
                switch (Key_Value)
                {
                case 3:       // Action corresponding to key 3.
                      ...
                      break;
                case 4:       // Action corresponding to key 4.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI Sending Function
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // Write Data to SPI sending register
    while (SPI_BUSY);    // Wait for Sending to complete
    RxDat = SPI_BUF;     // Get SPI received data
    update_key_status(RxDat); // Update library status
}
```

## 16-bit SPI

Same as the simplest application, but using a 16-bit SPI port (highlighted parts indicate the differences from the 8-bit mode):

```
In bc727x_config.h, should define this:
#define SPI_MODE_16BIT 1


#include “bc727x_lib/bc727x.h”
```

```
int main(void)
{
    set_write_func(spi_send);        // Set SPI sending function
    while(1)     // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // Get key value
                display_hex(Key_Value, 0, 2);    // Display key value
                switch (Key_Value)
                {
                case 3:      // Action corresponding to key 3.
                      ...
                      break;
                case 4:      // Action corresponding to key 4.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint16_t Dat)    // SPI sending function
{
    uint16_t RxDat;
    SPI_BUF = Dat;      // Write Data to SPI sending register
    while (SPI_BUSY);   // Wait for Sending to complete
    RxDat = SPI_BUF;    // Get SPI received data
    update_key_status(RxDat); // Update library status
}
```

## Using software simulated SPI

Same as the previous example, but without using the hardware SPI port and instead using software simulated SPI. (Highlighted parts show the differences from using hardware SPI)

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);        // Set SPI sending function
    while(1)     // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                display_hex(Key_Value, 0, 2);    // display key value
                switch (Key_Value)
                {
                case 3:      // Action corresponding to key 3.
                      ...
                      break;
                case 4:      // Action corresponding to key 4.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t i, RxDat;
    delay_half_T();     // delay for half clock period(>7.8us)
    for (i=0; i<8; i++)
```

```
    {
        if (Dat&0x80)        // Output MSB of Dat to MOSI
        {
            MOSI = 1;
        }
        else
        {
            MOSI = 0;
        }
        CLK = 0;
        delay_half_T();      // delay for half clock period(>7.8us)
        CLK = 1;
        delay_half_T();
        Dat = Dat<<1;        // shift Dat left for 1 bit
        RxDat == RxDat<<1;   // Read 1 bit to RxDat from MISO
        if (MISO)
        {
            RxDat = RxDat|0x01;
        }
    }
    update_key_status(RxDat); // Update library status
}
```

## Using CS

Based on the simplest application but added CS control.(highlighted parts indicate CS related operations):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);         // Set SPI sending function
    set_cs_high_func(cs_set_1);
    set_cs_low_func(cs_set_0);
    while(1)      // main loop
    {
        do_something();      // main task
        …
        if (is_key_changed())      // Query for new key press
        {
            Key_Value = get_key_value();      // get key value
            display_hex(Key_Value, 0, 2);     // display key value
            switch (Key_Value)
            {
            case 3:       // Action corresponding to key 3.
                ...
                break;
            case 4:       // Action corresponding to key 4.
                ...
                break;
            default:
                break;
            }
        }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // Write Data to SPI sending register
    while (SPI_BUSY);    // Wait for Sending to complete
    RxDat = SPI_BUF;     // Get SPI received data
    update_key_status(RxDat); // Update library status
}

void cs_set_1(void)
{
    CS = 1;
}

void cs_set_0(void)
{
```

```
    CS = 0;
}
```

## Controlling Multiple CS Lines

When the driver controls multiple BC727x chips at the same time (highlighted part indicates the added part compared to basic CS control):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    uint16_t     i, j;
    set_write_func(spi_send);        // Set SPI sending functio

    while(1)     // main task
    {
          use_cs1();                 // selecting 1st BC727x
          display_dec(i++, 0, 4);    // Display i with the 1st chip at position 0 with
a width of 4 digits
          use_cs2();                 // selecting 2nd BC727x
          display_dec(j--, 0, 5);    // Display j with the 2nd chip at position 0 with
a width of 5 digits
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                display_hex(Key_Value, 7, 2);    // display key value(still with 2nd
chip, at digit 7)
                switch (Key_Value)
                {
                case 3:      // Action corresponding to key 3.
                      ...
                      break;
                case 4:      // Action corresponding to key 4.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // Write Data to SPI sending register
    while (SPI_BUSY);   // Wait for Sending to complete
    RxDat = SPI_BUF;    // Get SPI received data
    update_key_status(RxDat); // Update library status
}

void use_cs1(void)      // change to CS1 control
{
    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;
}
```

## Using SPI Interrupt and CS

Modifying the basic application to use SPI interrupt mode and CS signal (highlighted code represents the changes)

```
In bc727x_config.h, include definition:
#define SPI_MODE_INTERRUPT 1

#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(write_spi);// Set SPI sending function
    set_cs_low_func(cs_set_0);        // set CS control function
    set_cs_high_func(cs_set_1);
    set_eni_func(enable_spi_int);    // set SPI interrupt enable function
    set_disi_func(disable_spi_int);
    while(1)      // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                CS = 0;
                display_hex(Key_Value, 0, 2);    // display key value
                CS = 1;
                switch (Key_Value)
                {
                case 3:       // Action corresponding to key 3.
                      ...
                      break;
                case 4:       // Action corresponding to key 4.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void write_spi(uint8_t Dat)    // SPI sending function
{
    SPI_BUF = Dat;       // wirte to SPI sending register
}

void cs_set_0(void)
{
    CS = 0;       // set CS to 0
}

void cs_set_1(void)
{
    CS = 1;       // set CS to 1
}

void enable_spi_int(void)
{
    SPI_IE = 1;   // enable SPI interrupt
```

```
}

void disable_spi_int(void)
{
    SPI_IE = 0;  // disable SPI interrupt
}

void SPI_ISR(void)      // SPI ISR, triggered when SPI sending is completed
{
    update_key_status(SPI_BUF);      // call update_key_status() with data received
    tx_ready();                      // tell library ready to send next data
}
```

## Detecting Key Release

Please refer to the following pseudocode (the highlighted part is the added part compared to the simplest application):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);        // Set SPI sending function
    set_detect_mode(1);        // enable key release detection
    while(1)     // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                display_hex(Key_Value, 0, 2);    // display key value
                switch (Key_Value)
                {
                case 0x03:   // Action corresponding to key 3.
                      ...
                      break;
                case 0x83:   // Action corresponding to key 3 release
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // Write Data to SPI sending register
    while (SPI_BUSY);   // Wait for Sending to complete
    RxDat = SPI_BUF;    // Get SPI received data
    update_key_status(RxDat); // Update library status
}
```

## Using Long-press  Keys – With Timer Interrupt

Please refer to the following pseudocode (highlighted parts indicate the added portions compared to the simplest application):

```
#include “bc727x_lib/bc727x.h”

const uint8_t LPDef[] = {2, 5}; // Long press key definition includes two keys: key 2 and key 5.

int main(void)
{
    set_write_func(spi_send);        // Set SPI sending function
    set_longpress_count(60);         // set long press time to 3s(50ms*60)
    def_longpress_key(LPDef, 2);     // define 2 long press keys
```

```
    while(1)      // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                switch (Key_Value)
                {
                case 3:       // Action corresponding to key 3.
                      ...
                      break;
                case 4:       // Action corresponding to key 4.
                      ...
                      break;
                case 0x22:    // Action corresponding to long press of key 2.
                      ...
                      break;
                case 0x25:    // Action corresponding to long press of key 5.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // Write Data to SPI sending register
    while (SPI_BUSY);    // Wait for Sending to complete
    RxDat = SPI_BUF;     // Get SPI received data
    update_key_status(RxDat); // Update library status
}

void TIMER_ISR(void)    // timer interrupt, served every 50ms
{
    long_press_tick();
}
```

## Using Long-press  Keys – Without Timer Interrupt

(Highlighted part shows the difference compared to timer method

```
#include “bc727x_lib/bc727x.h”

const uint8_t LPDef[] = {2, 5}; // Long press key definition includes two keys: key 2 and key 5.

int main(void)
{
    set_write_func(spi_send);         // Set SPI sending function
    set_longpress_count(1500);        // set long press time to 3s(2ms*1500)
    def_longpress_key(LPDef, 2);      // define 2 long press keys
    while(1)      // main loop
    {
          do_something();     // main task (assuming it cost 2ms)
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                switch (Key_Value)
                {
                case 3:       // Action corresponding to key 3.
                      ...
                      break;
                case 4:       // Action corresponding to key 4.
                      ...
                      break;
                case 0x22:    // Action corresponding to long press of key 2.
                      ...
                      break;
                case 0x25:    // Action corresponding to long press of key 5.
```

```
                        ...
                        break;
                default:
                        break;
                }
            }
            long_press_tick();
        }
    }

    void spi_send(uint8_t Dat)     // SPI sending function
    {
        uint8_t RxDat;
        SPI_BUF = Dat;      // Write Data to SPI sending register
        while (SPI_BUSY);   // Wait for Sending to complete
        RxDat = SPI_BUF;    // Get SPI received data
        update_key_status(RxDat); // Update library status
    }
```

## Using Combination Keys

Please refer to the following pseudocode (highlighted parts indicate additions compared to the simplest program)

```
    #include “bc727x_lib/bc727x.h”

    const uint16_t CBDef[] = {0x0003, 0x0c00}; // Define two combination keys: the first one is composed of key 0 and 1, while the second one is composed of key 10 and 11.

    int main(void)
    {
        set_write_func(spi_send);         // Set SPI sending function
        def_combined_key(CBDef, 2);       // define 2 combination keys
        while(1)      // main loop
        {
            do_something();     // main task
            …
            if (is_key_changed())      // Query for new key press
            {
                Key_Value = get_key_value();     // get key value
                switch (Key_Value)
                {
                case 3:       // Action corresponding to key 3.
                        ...
                        break;
                case 4:       // Action corresponding to key 4.
                        ...
                        break;
                case 0x10:    // Action corresponding to combination key 1.
                        ...
                        break;
                case 0x11:    // Action corresponding to combination key 2.
                        ...
                        break;
                default:
                        break;
                }
            }
        }
    }

    void spi_send(uint8_t Dat)     // SPI sending function
    {
        uint8_t RxDat;
        SPI_BUF = Dat;      // Write Data to SPI sending register
        while (SPI_BUSY);   // Wait for Sending to complete
        RxDat = SPI_BUF;    // Get SPI received data
        update_key_status(RxDat); // Update library status
    }
```

## Long-press of Combination Keys

(Highlighted part is related to long-press of the combination key

```
#include “bc727x_lib/bc727x.h”

const uint16_t CBDef[] = {0x0003, 0x0c00}; // Define two combination keys: the first one is composed of key 0 and 1, while the second one is composed of key 10 and 11.
const uint8_t LPDef[] = {0x10};            // define the combination key as a long press key(key value 0x10)


int main(void)
{
    set_write_func(spi_send);        // Set SPI sending function
    set_longpress_count(60);         // set long press time to 3s(50ms*60)
    def_longpress_key(LPDef, 1);     // define long press key
    def_combined_key(CBDef, 2);      // define combination key
    while(1)      // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();     // get key value
                switch (Key_Value)
                {
                case 3:      // Action corresponding to key 3.
                      ...
                      break;
                case 4:      // Action corresponding to key 4.
                      ...
                      break;
                case 0x30:   // Action corresponding to long press of the combination key.
                      ...
                      break;
                default:
                      break;
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // Write Data to SPI sending register
    while (SPI_BUSY);   // Wait for Sending to complete
    RxDat = SPI_BUF;    // Get SPI received data
    update_key_status(RxDat); // Update library status
}
```

## Using Callback Functions

Please refer to the following pseudocode (highlighted part is the relevant code):

```
#include “bc727x_lib/bc727x.h”

void key_action(unsigned char Key)      // Keyboard event processing function
{
      switch(Key)
      {
      case 2:      // Action corresponding to key 2.
            ...
            break;
      case 3:      // Action corresponding to key 3.
            ...
            break;
      default:
            break;
      }
}
```

```
int main(void)
{
    set_callback(key_action);          // set callback function
    set_write_func(spi_send);          // set SPI sending function
    while(1)      // main loop
    {
          do_something();      // main task
          …
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // Write Data to SPI sending register
    while (SPI_BUSY);    // Wait for Sending to complete
    RxDat = SPI_BUF;     // Get SPI received data
    update_key_status(RxDat); // Update library status
}
```

## Detect Keyboard Idle

Detecting long periods of no key press and not enabling key release detection, please refer to the following pseudocode (highlighted parts are related to 'no key press' detection)

```
#include “bc727x_lib/bc727x.h”

const uint8_t LPDef = {0x02, 0xff}; // define the long press key as key 2 and no key.

int main(void)
{
    set_write_func(spi_send);          // set SPI sending function
    set_longpress_count(60);           // set long press time to 3s(50ms*60)
    def_longpress_key(LPDef, 2);       // define 3 long press keys
    while(1)      // main loop
    {
          do_something();      // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();      // get key value
                switch (Key_Value)
                {
                case 3:       // Action corresponding to key 3.
                      ...
                      break;
                case 4:       // Action corresponding to key 4.
                      ...
                      break;
                case 0xff:    // ‘no key press’ event
                      no_key_time++;       // increase counter each time
                      if (no_key_time >= 10)     // it counter reaches 10 go to sleep
                      {
                            no_key_time = 0;
                            sleep();      // enter sleep
                      }
                      break;
                default:
                      break;
                }
                if (Key_Value != 0xff)     // If there’s other event
                {
                      no_key_time = 0;     // clear counter
                }
          }
    }
}

void spi_send(uint8_t Dat)     // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;       // Write Data to SPI sending register
    while (SPI_BUSY);    // Wait for Sending to complete
    RxDat = SPI_BUF;     // Get SPI received data
```

```
    update_key_status(RxDat); // Update library status
}

void TIMER_ISR(void)    // timer interrupt, served every 50ms
{
    long_press_tick();
}
```

## Get Timely Keyboard Using KEY Signal When Display Is Not Updated Frequently

The previous keyboard application examples assumed that the user program would update the display content periodically, with an interval of within 100ms. Because every operation of the SPI port will update the keyboard status, the driver library can understand the status change of the keyboard in a timely manner without checking the KEY pin. However, if the access interval of the program is relatively long, relying solely on passive acquisition of the keyboard mapping will result in a sluggish response and may also cause key loss. In this case, it is necessary to rely on detecting the KEY signal on the chip to understand the changes in the keyboard in a timely manner. Usually, the KEY pin is connected to an external interrupt. When the level of KEY changes, it will cause an interrupt, and in the interrupt handler, a pseudo-instruction 0xFF can be sent to force the SPI port to work and update the keyboard status to report the key event.

Because the driver library function uses the SPI port to send pseudo instructions in the external interrupt, if the interrupt occurs during the process of using the same SPI port in the main program, a resource conflict will occur. Therefore, if the KEY pin is used to trigger an external interrupt, the external interrupt should be temporarily disabled in the main program when performing BC727x operations; or in the service program of the external interrupt, before calling the function to send pseudo instructions, check whether the SPI is currently in use, and skip it if it is already in use (the purpose of sending a pseudo instruction is to force the SPI port to work to obtain the latest keyboard mapping, and if the SPI port is already in use, this operation is not necessary).

If the SPI port operation of the driver library is in interrupt mode, the `bc727x_send_cmd()` operation from the external interrupt may disturb the read/write of the SPI FIFO buffer. Fortunately, in interrupt mode, the user is required to provide functions to enable/disable SPI interrupts, and only needs to add the enable/disable of external interrupts in these two functions.

Pseudocode (highlighted part is the relevant code):

```
#include “bc727x_lib/bc727x.h”

int main(void)
{
    set_write_func(spi_send);        // set SPI sending function
    while(1)     // main loop
    {
          do_something();     // main task
          …
          if (is_key_changed())      // Query for new key press
          {
                Key_Value = get_key_value();      // get key value
                disable_EXTI();                   // disable external interrupt
                display_hex(Key_Value, 0, 2);     // display key value
                enable_EXTI();                    // enable external interrupt
                switch (Key_Value)
                {
```

```
                case 3:      // Action corresponding to key 3.
                       ...
                       break;
                case 4:      // Action corresponding to key 4.
                       ...
                       break;
                default:
                       break;
                }
            }
    }
}

void spi_send(uint8_t Dat)    // SPI sending function
{
    uint8_t RxDat;
    SPI_BUF = Dat;      // Write Data to SPI sending register
    while (SPI_BUSY);   // Wait for Sending to complete
    RxDat = SPI_BUF;    // Get SPI received data
    update_key_status(RxDat); // Update library status
}

void EXTI_ISR(void)     // EXT interrupt ISR, happens when KEY pin changes
{
    EXTIF = 0;          // clear interrupt flag
    send_cmd(0xff, 0xff);      // send pseudo-instruction
}
```