BC6xxx BC759x

UART Keyboard Interface

Arduino Library

User Manual

Index

Index....2
Overview....3
Library Installation....4
Hardware Connection....5
Usage....6
Create Instance....6
setup()....6
Basic Application: Single Key Events....7
Advanced Application: Long-press Keys....7
Advanced Application: Key Combinations....8
Function Reference Manual....10
Configuration Functions....10
setDetectMode() - set detection mode....10
setLongpressCount() - set long-press count....10
setCallback() : set callback functions....10
defCombinedKey() : define key combinations....11
Combination definination array....11
Key combination list array....11
defLongpressKey() - define long-press keys....12
Long-press key definition array....12
Long-press key list array....13
Input Functions....13
checkChanges() - update keyboard status....13
longpressTick() - long-press counter tick....14
Query Functions....14
isKeyChanged() - is keyboard status changed....14
getKeyValue() - get key value....14
Examples....16
Example1. Simple single key press....16
Example 2. Long-press key....16
Example 3. Long-press + combination keys....17

Overview

BC6xxx and BC759x series chips share a unified single line UART keyboard interface. This library can be used for the following chips:

BC6301 -- 30-key key matrix interface

BC6040 -- 40-key key matrix interface

BC6561 -- 56-key key matrix interface

BC6088 -- 88-key key matrix interface

BC7595 -- 48 segments LED driver and 48-key key matrix interface

BC7591 -- 256 segments LED driver and 96-key key matrix interface

This library is available for all Arduino devices, and can be used for both hardware and software serial ports.

By using this library, in addition to easily handling common single-key events, complex keyboard functions such as key combinations and long-press keys can be implemented with just a few lines of code.

In the UART single line keyboard interface, a keyboard event is represented by a single byte each time. Value 0-0x7F represents the key number value, the highest bit is used as a key release flag, that is, value 0-0x7F means the key is pressed, 0x80-0xFF means the key is released, so theoretically the maximum number of keys is 128. In reality the BC7591 can support up to 96 keys. This library uses the unused key number space to allow users to assign combination keys and long-press keys a custom key number, so that in the user program, handling of combination keys and long-press keys is as simple as the handling of ordinary keys, everything else is handled by the library.

The library is very simple to use, requiring only 3 steps.

1. In loop(), call the checkChanges() function.

2. query isKeyChanged()

3. If the result of isKeyChanged() is true, use getKeyValue() to get the key value

For detailed information about the UART keyboard interface, please refer to the datasheet of the BC6xxx or BC759x series chips.

Library Installation

The library is very easy to install, from the Arduino IDE menu, select

"Sketch --> Include Library --> Add .ZIP Library... "

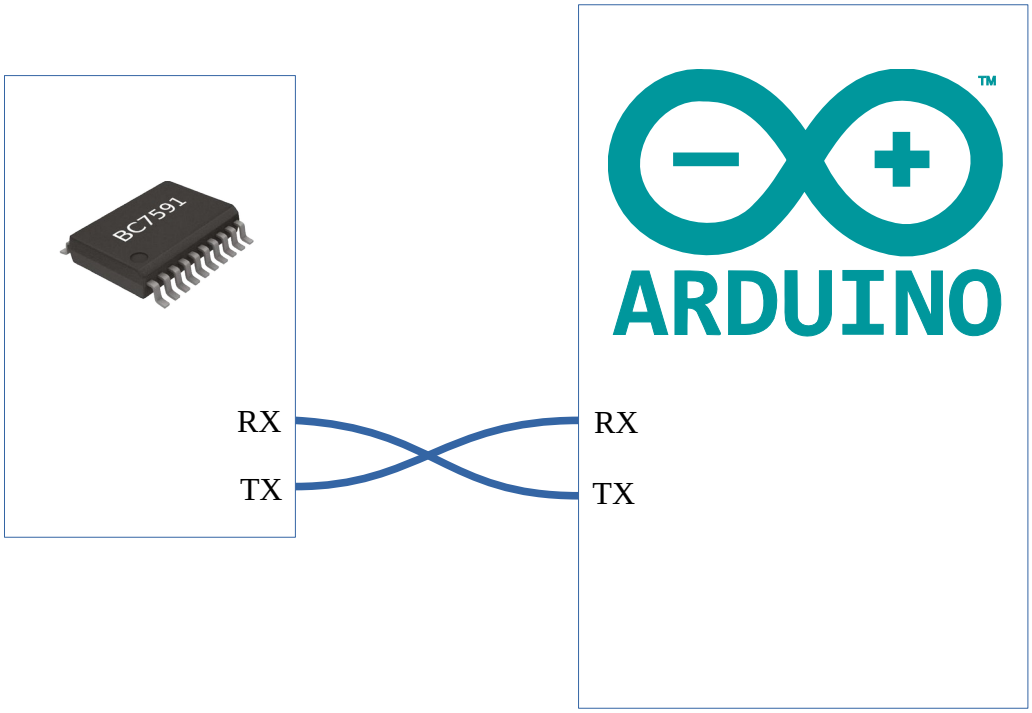
Choose BC_key_scan.zip , done!

Or directly unzip the downloaded BC_key_scan.zip file and manually copy the BC_key_scan directory to the libraries subdirectory in the Arduino directory to achieve the same effect.

After installation, you can find the BC_key_scan library in the "Project (Sketch) --> Include Library" menu, which means it is ready to use.

Hardware Connection

The BC6xxx and BC759x series chips both use UART serial communication, and the UART uses 2 pins TX and RX on the chip, where TX is the output of the chip's keyboard interface. The BC6xxx chip does not have RX pins, they are only available on the BC759x, which is used for the LED driver. When connected, the TX/RX pins of the chip are cross-connected with the RX/TX pins of the Arduino serial port



Usage

Create Instance

The interface of BC6xxx and BC759x series chips is serial port, so the use of this library must also depend on the serial port of Arduino. The serial port can be Arduino's hardware serial port (such as Serial, Serial1, Serial2, etc.) or software serial port (Software Serial).

For models like the Arduino UNO, there is only one hardware serial port, and that serial port is already used for communication with the computer, so it is better to use the software serial port, otherwise there will be conflicts with the IDE (unless the hardware serial port is not used).

Before use, in addition to loading the library, an instance of the library must be created, e.g.

```
BcKeyScan       Keypad(Serial1);
```

Where `BcKeyScan` is the name of this driver, `Keypad` is the name given to the instance by the user, it can be anything, as long as it conforms to the Arduino variable naming rules. The serial port in parentheses is the serial port used, if it is a hardware serial port, it should be `Serial`, or `Serial1`, `Serial2`, etc.

If you are using the software serial port, you need 2 extra steps. The first step requires loading the software serial library `SoftwareSerial`, which is one of the standard libraries for Arduino. The second step requires that an instance of the software serial port is created before an instance of this keyboard driver is created:

```
SoftwareSerial swSerial(2, 3);
```

Where `swSerial` is the name given to the instance by the user, which can be anything that matches the Arduino variable name naming rules. In the brackets `2, 3` are the RX and TX pins used by the software serial port. See the Arduino software serial port library for more information.

The instance of this library can be created only after the instance of the software serial port is created:

```
BcKeyScan       Keypad(swSerial);
```

setup()

This driver library itself does not require any initialization, however the serial port must be properly initialized to properly connect to the BC6xxx and BC759x chips. The serial port must be initialized to 9600 baud rate.

```
Serial1.begin(9600);        // For hardware serial port
```

or

```
swSerial.begin(9600);       // For software serial port
```

Although the library can be used without initialization, however, if the user needs to use features like key combinations or long-press keys, some additional steps will be required and there are settings that need to be done in setup(). See later for more details.

Basic Application: Single Key Events

Once the keyboard interface instance is created, only three lines of code are needed to implement the basic keyboard functions.

1. In loop(), call the checkChanges() function to update the status of the keyboard.

2. Call the isKeyChanged() function, which returns a bool value of true or false; if the return value is true, there is a new key event, if false, there is no event to be processed.

3. If the return value of the previous step is true, the getKeyValue() function should be called and the return value of the function is the key value of the key that changed.

Knowing which key was pressed, the next step is just to turn to the appropriate processing as needed. In the default state, the library only reports the key press event and ignores the key release. However, both the BC6xxx and BC759x chips have the ability to detect key presses and releases separately, so if the user wants to detect the release of a key, for example, to have the key pressed and released perform different actions, the library can be set to enable the detection of the key release:

```
setDetectMode(1);          // Set the library to detect the key release too.
```

There are currently 2 detection modes, 0 is the default mode, which does not detect key release, and 1 is to detect both key presses and releases. The key detection mode can be changed at any time, if you don't need to change the key detection mode during the whole application, you can put this setting operation into the setup() function

Advanced Application: Long-press Keys

The library assigns key values to long-press keys as normal single key press, so that the user can handle long-press keys as normal single keys. The key values of the long keys are defined by the user, and the defined key values should be chosen from unused values so that they do not conflict with the normal keys. The valid key values of the UART keyboard interface protocol range from 0 to 127, and the maximum number of keys of the current BC6xxx and BC759x chips is 96 keys. Each chip has its natural key values start from 0 and end at N-1, where N is the number of keys that chip can support.

To use long-press keys, you first need to define which keys need to be detected as long-press keys and the value assigned to each long-press key by the user and inform the library. The definition of long-press keys is done through two arrays: "Long-press key definition" and "Long-ress key list". The first array, which defines the custom key value for the long-press key, is of 'unsigned char' type and consists of two elements, the first one is the key value of the key that needs to be detected the

long-press key, and the second one is the key value assigned by the user to represent the long key press. You can define as many long-press keys as you need. An example is as follows.

```
const unsigned char Lp1[2] = {5, 120};
const unsigned char Lp2[2] = {20, 121};
```

The above definition indicates that the custom key value for the long-press of key number 5 is defined as 120; the custom key value for the long-press event of key number 20 is defined as 121.

After defining the key values for the long-press keys, we need another array to inform the library about these definitions. This second array is actually a list of those custom key definition arrays above.

```
const unsigned char* LPList[2] = {Lp1, Lp2};
```

With this second array, the function defLongpressKey() can be used to tell the library to start detecting long press for these keys.

`defLongpressKey(LPList, 2);` // The second parameter "2" is the number of elements in the array LPList[]

This function, in general, can be called in `setup()`. After these steps, the library can now detect long keystrokes, but there is one issue that is not covered, which is how long the long keystrokes are actually pressed can be treated as a long-press.

There is no timing program inside the library itself, so the long press time is not calculated on a time basis, but indirectly through an internal counter that get the time by the count value. The user needs to call the `longpressTick()` function periodically, and each time it is called, the internal counter is incremented by one, and when the count value reaches the preset threshold, the long press time is considered to have been reached. The user needs to determine the time interval between calls to the `longpressTick()` function and then calculate the upper limit of the count based on the desired long press time.

The default value of the upper limit of the count is 200, but the user can change it with the `setLongpressCount()` function, and the upper limit is in the range of 1-65534.

Usually `setLongpressCount()` will be placed in `setup()`, while the `longpressTick()` function , is called in `loop()`.

See later for an example of actual usage.

Advanced Application: Key Combinations

The use of key combinations is very similar to long-press keys, and requires two arrays- "key combination definition" and "key combination list", to tell the library how to detect key combinations, except that the format of the "key combination definition" array is different from that of long-press keys.

The data type of the key combination definition array is also "unsigned char", but the length of the key combination definition array is determined by the number of keys that make up the key combination and is variable. For example, in the following two definitions:

```
const unsigned char Cb1[] = {2, 122, 0, 7};
const unsigned char Cb2[] = {3, 123, 11, 15, 20};
```

The first element is the number of elements that make up the key combination, i.e., the number of keys, and the second value is a user-defined key value representing the key combination, followed by a variable-length list of key values representing the keys that make up the key combination. In the above two examples, Cb1 consists of 2 keys, with a custom key value of 122, consisting of keys 0 and 7. Cb2 consists of 3 keys, with custom key value 123, consisting of keys 11, 15, and 20.

The key combination can consists of up to 7 keys.

The key combination list array format is the same as the long-press key list.

```
const unsigned char* CBList[] = {Cb1, Cb2};
```

Once you have the list of combined keys, call the `defCombinedKey()` function, which is used in the same way as the `defLongpressKey()` function.

```
defCombinedKey(CBList, 2);
```

This function is also generally called in `setup()`.

Once the key combination is defined, no additional operations are required to work. When the defined key combination occurs, such as in the above example when keys 0 and 7 are pressed at the same time, `isKeyChanged()` will return true, and `getKeyValue()` will return 122.

Function Reference Manual

Configuration Functions

setDetectMode() - set detection mode

Use format:

```
setDetectMode(Mode);
```

This function controls the detection mode of the library. When the input parameter `Mode` is 0, this library will only report the key press (on) event and ignore the key release; when `Mode` is 1, the library will detect both the key press and release, and a key press and release will both make the result of `is_key_changed()` query not 0(true). Here the key also includes the user-defined key combination.

The default operating state of the is `Mode`=0, which does not detect key release.

setLongpressCount() - set long-press count

Use format:

```
setLongpressCount(CountLimit);
```

The library's long-press key detection time is achieved by counting the number of times the `longpressTick()` function has been called. For keys set to detect long presses, if the number of times `longpressTick()` is called exceeds the `CountLimit` value set here, and the key remains in the current state during this time, the library reports a long press event.

The default value of the long-press count is 200, the valid range is 1-65534.

setCallback() : set callback functions

Use Format:

```
set_callback(*pCallbackFunc);
```

The input parameter of this function is a function pointer, which has unsigned char input parameter type and void return type. When this callback function is set, the function as the parameter will be called automatically whenever a new key event is generated, and the `isKeyChanged()` query will no longer be able to query for new key events. When the callback function is called, it will take the key value of the new key event as an argument, and the user can complete the key processing within the callback function. Callback function also works with user defined the keys, such as key combinations and long-press keys.

The callback function provides the user with another way to handle keystrokes other than the "`isKeyChanged() -- get_key_value()`" method. The callback function automatically checks if there is a new key event every time `checkChanges()` is called, and if there is, it will automatically go to the callback function. `checkChanges()`can be called in `serialEvent()`,

so that the key event is automatically handled as soon as it occurs. This function is for advanced users.

defCombinedKey() : define key combinations

Use format:

```
defCombinedKey(pCBKeyList, CBKeyCount);
```

If you need to use a key combination, you must inform the library of the definition of the key combination through this function. The way this library works with key combinations is that the user defines a specific keyboard combination and assigns a special key value to it. The library then monitors the keyboard for this specific combination and reports a new key event when the combination appears, just as it does with normal keys.

The definition of the key combination is done by means of 2 (kind of) arrays.

Combination defininationarray

Defines which keys each key combination consists of. The data type of the array is "unsigned char" and the contents are in the following format: `{count, defined_value, key1, key2, ... keyn}` where `count` is the number of keys in the key combination, 2 for 2 keys and 3 for 3 keys. A key combination can consist of up to 7 keys. `defined_value` is a special key value assigned to the key combination by the user, when the key combination occurs, the library will report a new key event and the key value is the `defined_value`. To avoid conflicts with individual key values, this custom key value should avoid the key values already used by the chip itself. User defined values are usually in the range of 97-126. `key1, key2 ... keyn`, are the original key values of the keys that make up the key combination.

A real world example:

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

The definition above indicates that the key combination named `cb1` consists of two keys, key 3 and key 19, and is assigned a special key value of 125; the key combination named `cb2` consists of three keys, key 4, key 7 and key 10, and is assigned a special key value of 124.

This array is not used directly in calls to the `defCombinedKey()` function, but is a member of the following array of combined key lists and is therefore required.

Key combination list array

The array is of type "unsigned char*" , and its members are a list of key combination definitions. The array has as many members as combination keys in total. Take the above key combination definition as an example, there are two key combinations `cb1` and `cb2`, the definition of the key combination list array will be

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

With the definition of the above 2 arrays, you have all the parameters to call the `defCombinedKey()` function, still with the above data as an example, the way to call this function is:

```
defCombinedKey( cb_list, 2);
```

`Input parameters:`

`pCBKeyList` - a pointer towards combination key list array

`CBKeyCount` - number of combination keys

The key combination should only be defined once in `setup()` and should not be defined repeatedly.

defLongpressKey() - define long-press keys

Use format:

```
defLongpressKey(pLPKeyList, LPKeyCount);
```

Defining a long-press key is very similar to defining a key combination. The function has two input parameters, which are:

`pLPKeyList` : a pointer towards long-press key list

`LPKeyCount` : number of long-press keys

The 2 arrays required for passing the long-press key definition:

Long-press key definition array

Defines which key the long-press key is and the user-defined key value that represents the long press of that key. The array is of type "unsigned char", with a fixed length of 2 bytes, and has the format `{key_value, defined_value}`. Where `key_value` is the original key value of the key to be detected as a long-press key. This `key_value` can also be a user-defined key value of a key combination, means the detection of long-press of a key combination is also possible. `defined_value` is the user-defined key value representing the long press of the key. The selection of the custom key value follows the same principle for the custom key value of the key combinations, you should choose to use a key value that will not conflict with other keys.

Each key that needs to detect a long press corresponds to an array of long key definitions. If the system needs 2 long-press keys, then 2 long-press key definition arrays need to be defined. The following example is given:

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

In the above definition, two long-press keys are defined, the first one is key 4, which is an individual physical key on the keyboard with a custom long-press key value of 122, i.e.

when the key is pressed for a long time, the library will report a new key event with a key value of 122. The second key to be detected is a user-defined key combination with a key value of 124 and a custom key value of 121 for the long-press. The implication is that if a user-defined key combination with a key value of 124 is pressed for a certain amount of time, the library will report a new key event with a key value of 121. If we follow the previous example, it means that when the physical keys 4, 7 and 10 are pressed simultaneously and held for a certain amount of time, a new key event with a value of 121 will be generated.

Long-press key list array

The array type is "unsigned char*" and its content is a pointer to the long-press key definition array. The number of elements of the array is the number of long-press keys in the system. Using the above example, the resulting list array would be

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

Once you have the arrays defining the long-press keys, you can call this function to inform the library of the long-press keys that need to be detected:

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

The meaning of the above function is: use the long-press keys in the list `lp_list`, the total number of keys is 2.

When defining a long-press key, there is a special key value that is used exclusively to represent a no-keypress operation. If the original key value is defined as 255 (0xFF), it represents the detection of a "no key press". If there is no key press after the last key release for a preset long key press time, the library reports a key event with the custom key value set here. An example of a defined array to detect "no key press":

```
const unsigned char no_key[] = {255, 255};
const unsigned char* lp_list[] = {lp1, lp2 ... no_key};
```

In the above example, the custom key value for "no key press" detection is also set to 255. With long key press detection enabled, the library will report a key press event with a value of 255 if no key press is made after the last key press has been released for the preset "long key press time".

For "no key" detection, it is not necessary to enable the key release detection mode.

Input Functions

checkChanges() - update keyboard status

Use format:

```
checkChanges();
```

This function is used to check if a new key has been pressed. After this function is called, if the keyboard status has been changed since the last time this function is called, the library will report a

new keyboard event and `isKeyChanged()` will return true. If it is set to use a callback function, the callback function will be called before this function returns.

longpressTick() - long-press counter tick

Use format:

```
longpressTick();
```

The detection of long key press depends on this function. This function should be called periodically. Each time this function is called, the counter inside the library is increased one step, and each time there is a new keyboard activity, this counter is cleared. If this counter reaches the set upper limit (set by the `setLongpressCount()` function), it will check if the last key press was one of the long presses to be detected, and if so, a long press event will be reported.

If the time interval for calling this function is fixed, then the time for a long key press will be the call interval multiplied by the set count limit.

This function can generally be called in `loop()`, and if the user needs to temporarily stop the detection of a long keystroke, he only needs to temporarily stop calling this function.

Query Functions

isKeyChanged() - is keyboard status changed

Use format:

```
isKeyChanged();
```

The user program uses this function to get information about the status of the keyboard. The return value of the program is of type bool, when it returns false (0), it means that there are no new key changes, and when it returns true (1), it means that there is a new key(change). A new key is a key change that can be detected in the current working mode, for example if it is set to "not detect key release", any key release event will not affect the result of this query. New key(change) also include both key combination and long keystroke events.

For combination keys, when each key in the combination member is pressed, a new key event for the individual key is reported as if it were a separate key, but when the last key in the combination is pressed, only the new key event for the combination key is reported because the combination key condition has been satisfied, and no key events for that individual key are generated separately.

When there are two keys pressed at the same time, two key events will be generated consecutively, but if the program fails to get the key value with `getKeyValue()` in time after the first key event is generated, the event will be replaced by the second keyboard event. If the first key is an important key to be captured, it is recommended to set it and the second key as a key combination.

getKeyValue() - get key value

Use format:

`getKeyValue();`

The user gets the key value of the keyboard event with this function. This function can be called at any time. After this function is called, if the `isKeyChanged()` function returned true before, (which means there was a new key value that had not been acquired), `isKeyChanged()` will reverts to returning false.

The library provides the function of key latching, the latest key value can be read by this function until the a new key is pressed, thus reducing the requirement for the user program to process the key events in real-time. When set to not detect key release, the time interval between two consecutive key presses is usually at least a few hundred milliseconds, allows the user program to have sufficient time to process the current task and then react to the keyboard when it is free. If set to detect both key presses and releases, both key presses and releases generate keyboard events, and this time interval may be reduced to a few tens of milliseconds.

If a callback function is used to handle a key event, there is no need to call this function since the key value is already passed as an argument to the callback function.

Examples

Example1. Simple single key press

The following Sketch uses the software serial port, using the digital I/O pin 11 as RX.(Also set pin 12 as TX, because the software serial port initialization needs to set both RX and TX, but TX is not used in this example) The software monitors the key changes, and if there is a new key, the key value is printed out on the hardware serial port, which can be viewed with the Serial Monitor) to view.

```
#include <SoftwareSerial.h>
#include <bc_key_scan.h>

SoftwareSerial      swSerial(11, 12);   // Create software serial port instance with RX on pin 11 and TX on pin 12
BcKeyScan           Keypad(swSerial);   // Create a library instance, using the software serial port

void setup() {
      Serial.begin(9600);               // hardware serial port initialization
      swSerial.begin(9600);             // software serial port initialization
}

void loop() {
      Keypad.checkChanges();                  // update keyboard status
      if (Keypad.isKeyChanged() == true) {    // if there is new key
            Serial.println(Keypad.getKeyValue());   // print key value at hardware serial
      }
      delay(10);                        // delay for 10ms
}
```

Example 2. Long-press key

The following Sketch is basically the same as Example 1, with the addition of support for long-press keys.

```
#include <SoftwareSerial.h>
#include <bc_key_scan.h>

SoftwareSerial      swSerial(11, 12);   // Create software serial port instance with RX on pin 11 and TX on pin 12
BcKeyScan           Keypad(swSerial);   // Create a library instance, using the software serial port
```

```
// 定义长按键
const unsigned char  lp1[2] = { 1, 120 }; // Long-press key 1 definition: detection of '1' key long press, custom key value 120
const unsigned char  lp2[2] = { 5, 121 };  // Long-press key 2 definition: detection of '5' key long press, custom key value 121
const unsigned char* LPList[2] = { lp1, lp2 };  // long-press key list

void setup() {
	Serial.begin(9600);		// hardware serial port initialization
	swSerial.begin(9600);		// software serial port initialization
	Keypad.defLongpressKey(LPList, 2);	// definition of long-press keys
	Keypad.setLongpressCount(300);	// define the long-press time to 3s(10ms*300)
}

void loop() {
	Keypad.checkChanges();			// update keyboard status
	if (Keypad.isKeyChanged() == true) {	// if the keyboard status changed
		Serial.println(Keypad.getKeyValue());	// print key value at hardware serial
	}
	Keypad.longpressTick();		// increase the long-press counter
	delay(10);			// delay for 10ms
}
```

Example 3. Long-press + combination keys

Based on example 2, added key combinations.

```
#include <SoftwareSerial.h>
#include <bc_key_scan.h>

SoftwareSerial	swSerial(11, 12);
BcKeyScan	Keypad(swSerial);

// defining long-press keys
const unsigned char  lp1[2] = { 1, 120 };
const unsigned char  lp2[2] = { 5, 121 };
const unsigned char* LPList[2] = { lp1, lp2 };

// defining key combinations
const unsigned char  cb1[4] = { 2, 122, 0, 1 };  // Key combination 1 composed by key0 and key1 two keys, custom key value 122
```

```
const unsigned char  cb2[4] = { 2, 123, 8, 12 }; // Key combination 2 composed by key8 and key12 two keys, custom key value 123
const unsigned char* CBList[2] = { cb1, cb2 };    // key combination list


void setup() {
	Serial.begin(9600);
	swSerial.begin(9600);
	Keypad.defLongpressKey(LPList, 2);
	Keypad.setLongpressCount(300);
	Keypad.defCombinedKey(CBList, 2);  // defining key combinations
}

void loop() {
	Keypad.checkChanges();
	if (Keypad.isKeyChanged() == true) {
		Serial.println(Keypad.getKeyValue());
	}
	Keypad.longpressTick();
	delay(10);
}
```