3. Key control buzzer sounding

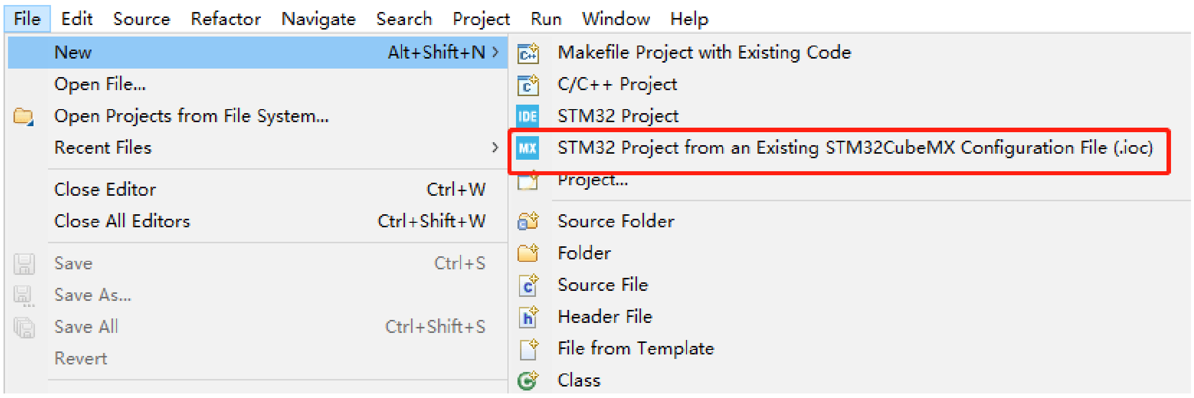
3.1. Purpose of Experiment

Detect the status of KEY1 on the expansion board and control the buzzer to sound. Every time you press a key, the buzzer sounds once.

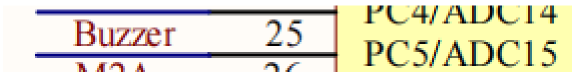
3.2 Configuring Pin Information

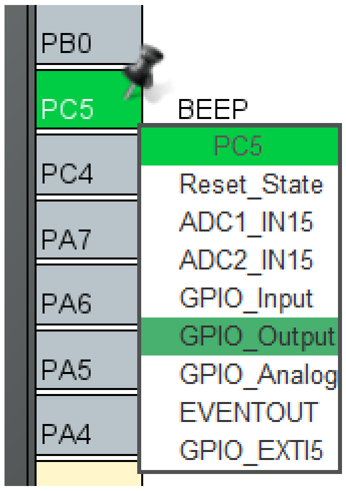
As we need to configure the information every time we create a new project, it is quite troublesome, good thing STM32CubeIDE provides the function of importing .ioc file, which can help us save time.

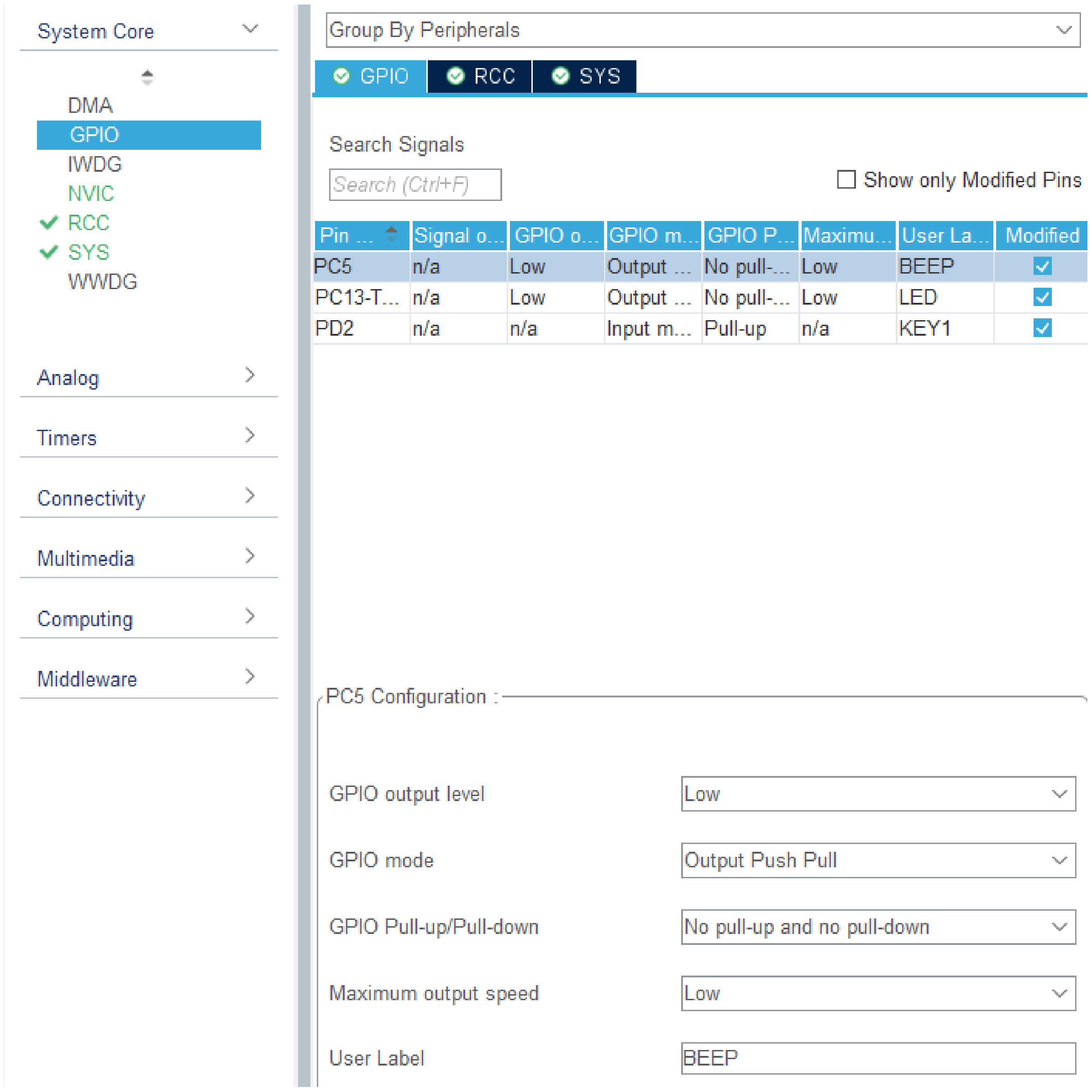
1. Import the ioc file from the LED's project and name it BEEP.



2. According to the schematic diagram, the control pin of the buzzer is connected to the PC5 pin of the STM32 chip. You need to set PC5 to GPIO_Output mode and modify the label to BEEP, and other configurations are shown in the figure below.





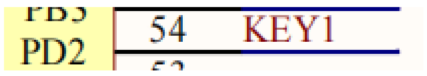


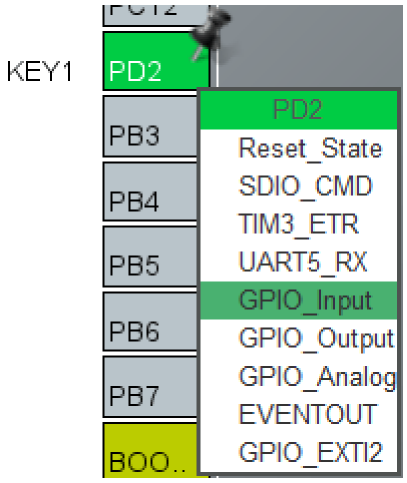
Pin ...	Signal o...	GPIO o...	GPIO m...	GPIO P...	Maximu...	User La...	Modified
PC5	n/a	Low	Output ...	No pull-...	Low	BEEP	✓
PC13-T...	n/a	Low	Output ...	No pull-...	Low	LED	✓
PD2	n/a	n/a	Input m...	Pull-up	n/a	KEY1	✓

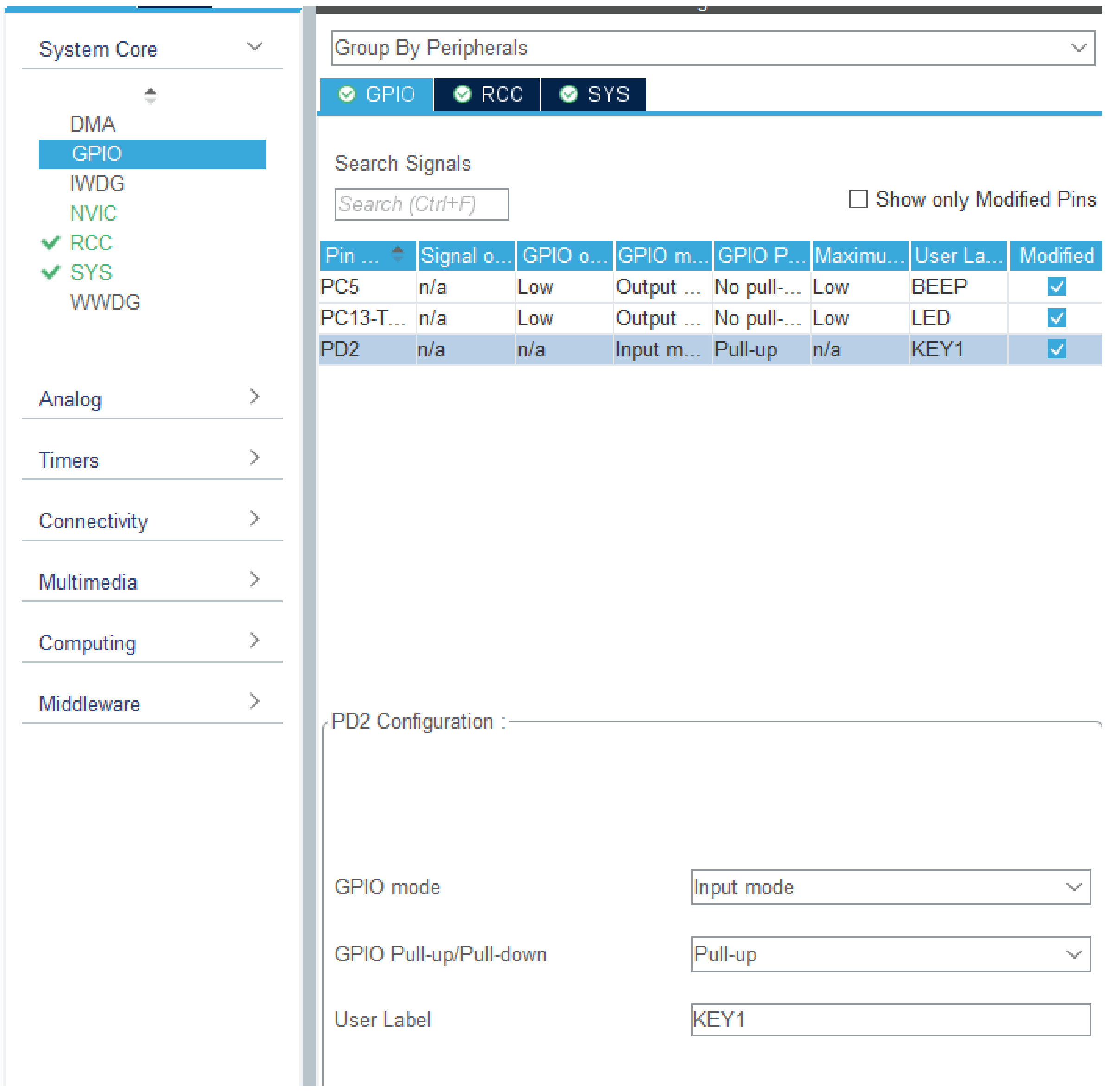
PC5 Configuration :

GPIO output level	Low
GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	No pull-up and no pull-down
Maximum output speed	Low
User Label	BEEP

3. Key KEY1 is connected to the PD2 pin. You need to set PD2 to GPIO_Input mode and change the label to KEY1, other configurations are shown below.







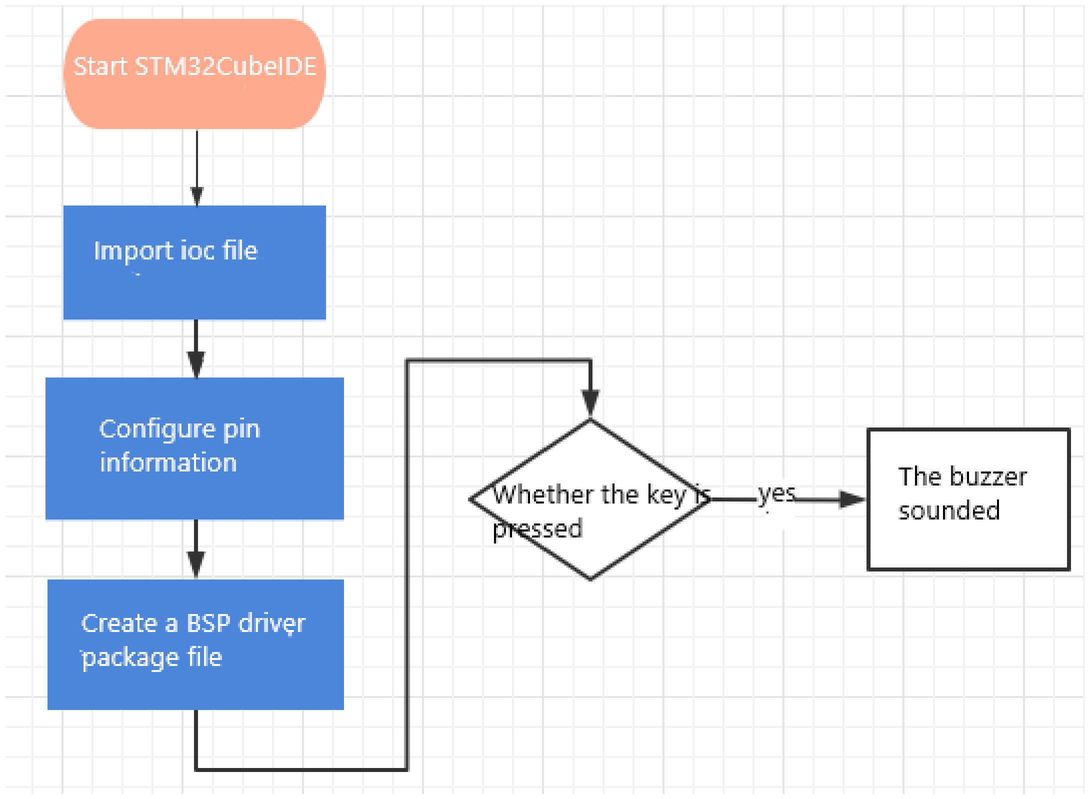
Pin ...	Signal o...	GPIO o...	GPIO m...	GPIO P...	Maximu...	User La...	Modified
PC5	n/a	Low	Output ...	No pull-...	Low	BEEP	✓
PC13-T...	n/a	Low	Output ...	No pull-...	Low	LED	✓
PD2	n/a	n/a	Input m...	Pull-up	n/a	KEY1	✓

PD2 Configuration :

GPIO mode	Input mode
GPIO Pull-up/Pull-down	Pull-up
User Label	KEY1

Save and generate the code.

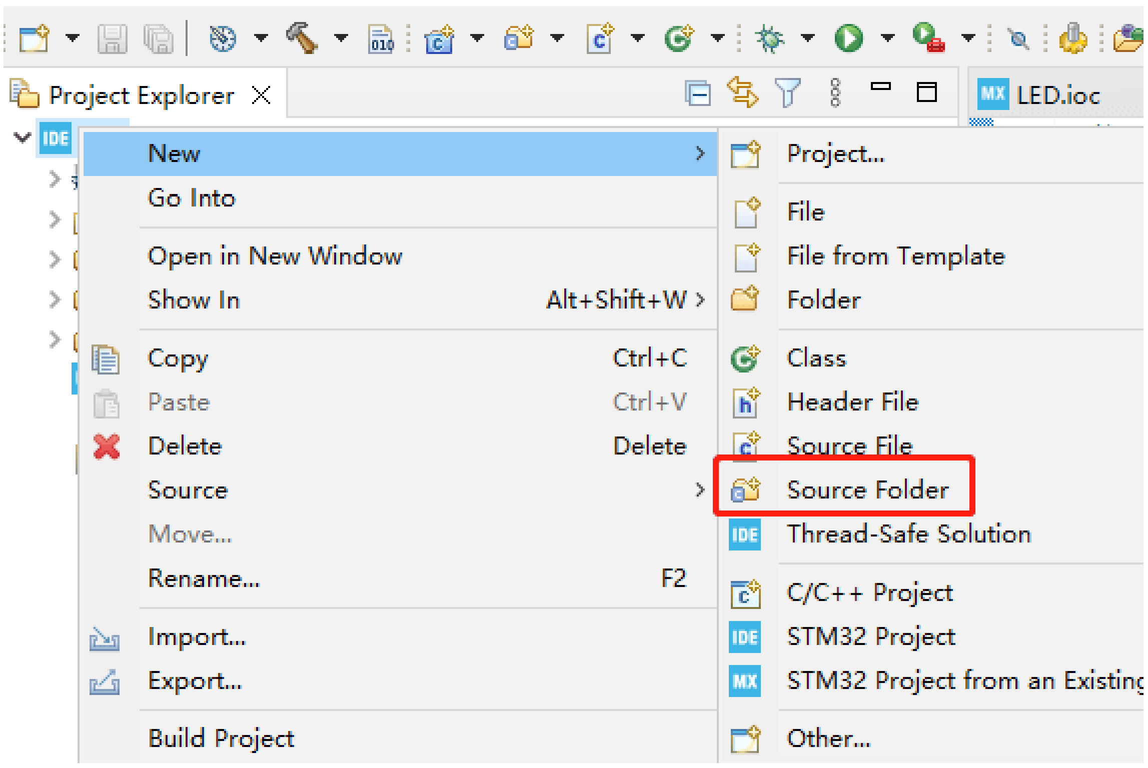
3.3. Experimental flow chart analysis

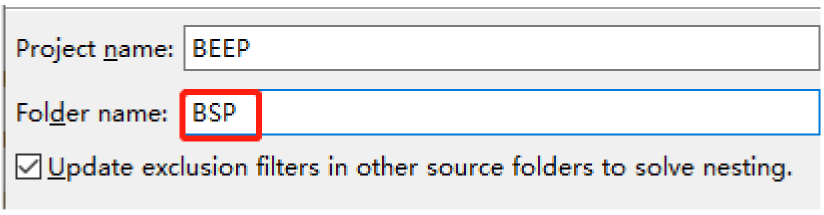

Start STM32CubeIDE
Import ioc file
Configure pin information
Create a BSP driver package file
Whether the key is pressed
yes
The buzzer sounded

3.4 Adding File Structure

The pin information has been configured graphically, and the generated code already contains the system initialization content, so there is no need to initialize the system configuration additionally.

1. For management convenience, we create a new BSP source code folder. In the mouse over the name of the project where the right click -> New -> Source Folder



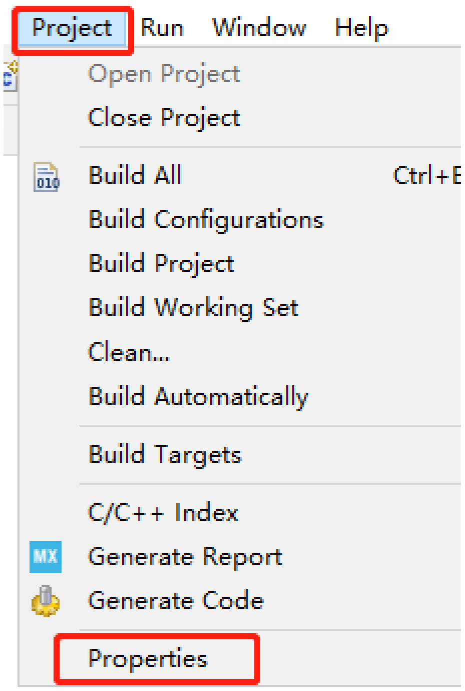


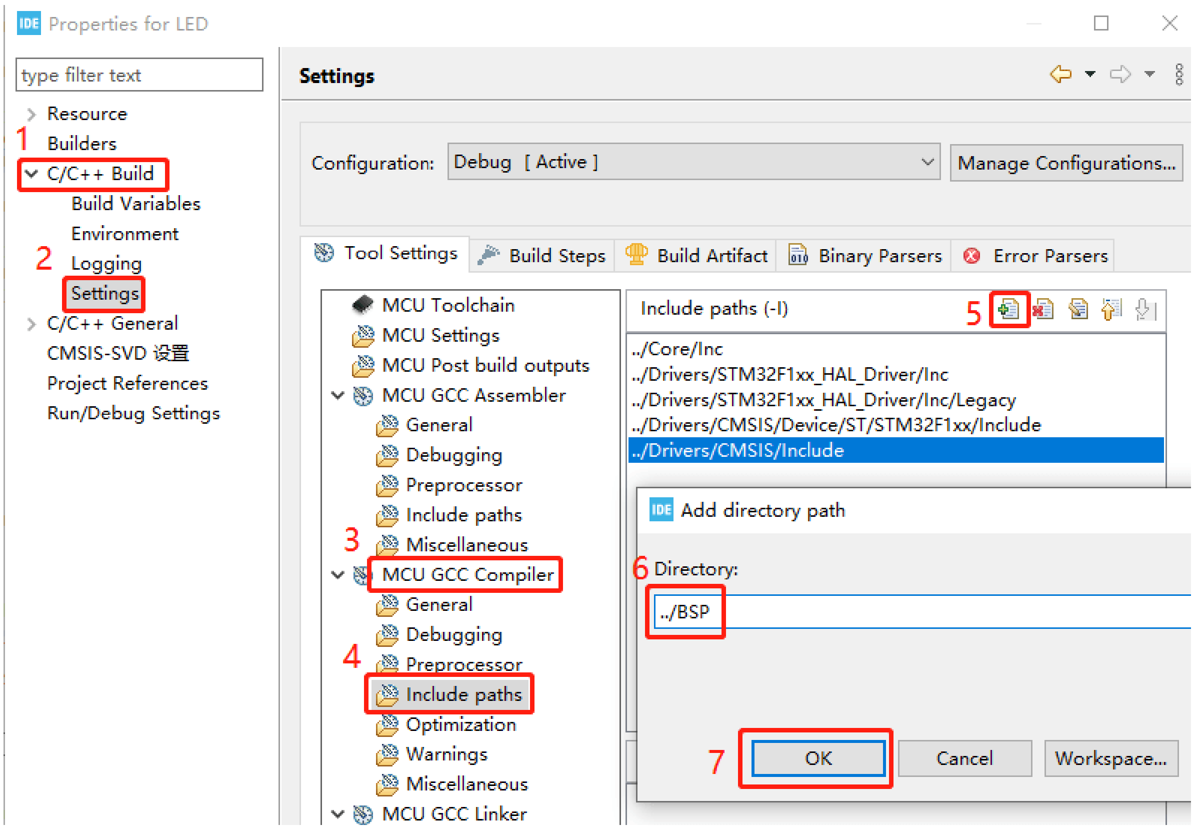
Project name: BEEP

Folder name: BSP

☑ Update exclusion filters in other source folders to solve nesting.

2. Add the BSP to the environment. Click Project->Properties->C/C++ Build->settings->MCU GCC Compiler->include paths, and then click the Add button to add the... /BSP in and save it.





Configuration: Debug [Active]

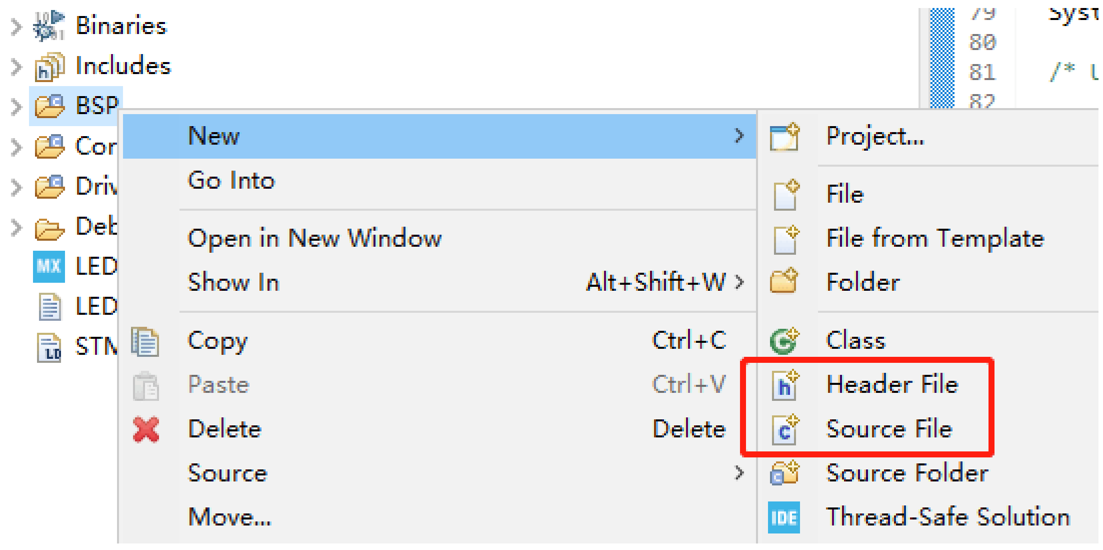
Include paths (-I)

../Core/Inc
../Drivers/STM32F1xx_HAL_Driver/Inc
../Drivers/STM32F1xx_HAL_Driver/Inc/Legacy
../Drivers/CMSIS/Device/ST/STM32F1xx/Include
../Drivers/CMSIS/Include

Directory: ../BSP

3. Create a new bsp.h and a bsp.c file, right-click BSP->New->Header File/Source File, and then enter the corresponding name can be.

These two files are mainly responsible for linking the main.c part of the function, you can avoid duplication of code.



4. in bsp.h add the following: the control of the LED into a macro definition of the way, simple and fast. New Bsp_Init () function is mainly responsible for the initialization, Bsp_Loop () is mainly responsible for the content of the main program. Bsp_Led_Show_State_Handle () function is mainly responsible for the LED indicator flashing effect, used to prompt the system is running.

```
/* DEFINE */
#define LED_ON()          HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, SET)
#define LED_OFF()         HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, RESET)
#define LED_TOGGLE()      HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin)


/* functions */
void Bsp_Init(void);
void Bsp_Loop(void);
void Bsp_Led_Show_State_Handle(void);
```

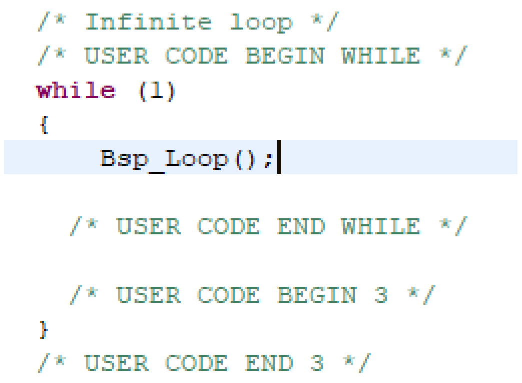
5. In the main.c file import bsp.h header file.

```
/* Private includes -------------
/* USER CODE BEGIN Includes */
#include "bsp.h"

/* USER CODE END Includes */
```

6. Call Bsp_Init() in the main function.

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
/* USER CODE BEGIN 2 */
Bsp_Init();
/* USER CODE END 2 */
```

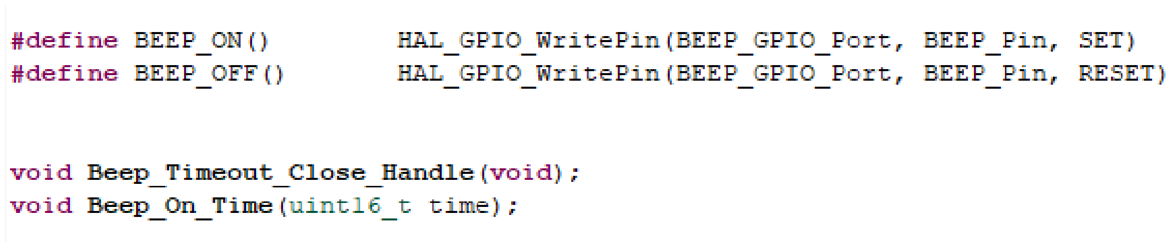
7. Call Bsp_Loop() in while(1).



```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    Bsp_Loop();

  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

3.5 The core code explanation

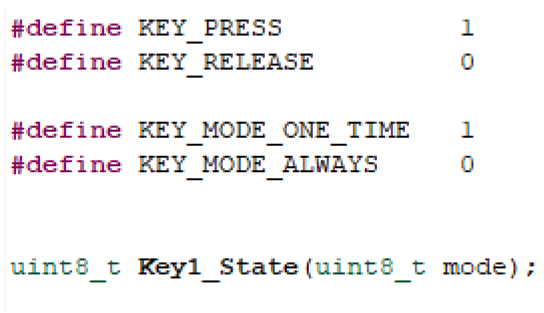
1. Create a new buzzer driver library bsp_beep.h and bsp_beep.c file in BSP. Add the following to bsp_beep.h:



```
#define BEEP_ON()          HAL_GPIO_WritePin(BEEP_GPIO_Port, BEEP_Pin, SET)
#define BEEP_OFF()         HAL_GPIO_WritePin(BEEP_GPIO_Port, BEEP_Pin, RESET)


void Beep_Timeout_Close_Handle(void);
void Beep_On_Time(uint16_t time);
```

Where Beep_Timeout_Close_Handle() function needs to be called every 10 milliseconds, so as to ensure that the Beep_On_Time() function sets the time after the normal effect in accordance with the Beep_On_Time() function. time in Beep_On_Time(time) represents the time when the buzzer is turned on, if time=0 then the buzzer is turned off If time=0, the buzzer will be turned off, if time=1, the buzzer will be on all the time, if time>=10, the buzzer will be turned off automatically after time milliseconds (time should be a multiple of 10).

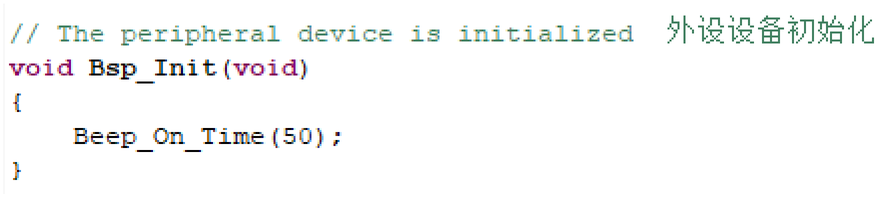
2. In the BSP in the new buzzer driver library bsp_key.h and bsp_key.c file. In bsp_key.h add the following content:



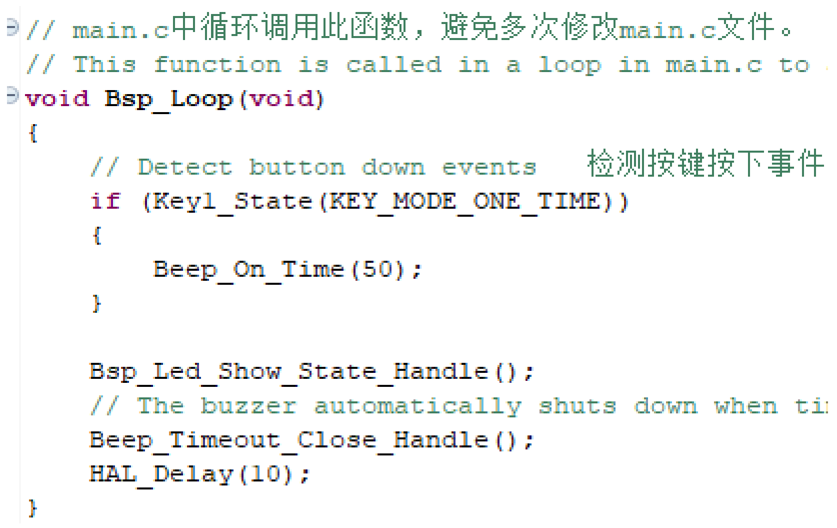
```
#define KEY_PRESS          1
#define KEY_RELEASE        0

#define KEY_MODE_ONE_TIME  1
#define KEY_MODE_ALWAYS    0


uint8_t Key1_State(uint8_t mode);
```

The function of Key1_State(mode) is to detect whether the key is pressed or not, and it needs to be called once every 10 milliseconds. mode can be entered as 0 or 1, mode=0 means that pressing KEY1 always returns KEY_PRESS, and releasing it only returns KEY_RELEASE, and mode=1 means that no matter how long KEY1 has been pressed, it will return KEY_PRESS once, and in all other cases, it will return KEY_RELEASE. mode=1 means that no matter how long KEY1 is pressed, it will only return KEY_PRESS once, and in all other cases, it will return KEY_RELEASE.

3. In Bsp_Init(), add power-on buzzer for 50 ms, and in Bsp_Loop(), detect whether the key is pressed or not, if it is pressed, it will sound for 50 ms and then turn off automatically. At the bottom are the control handles for the LEDs and buzzer, which only need to be called every 10 milliseconds.



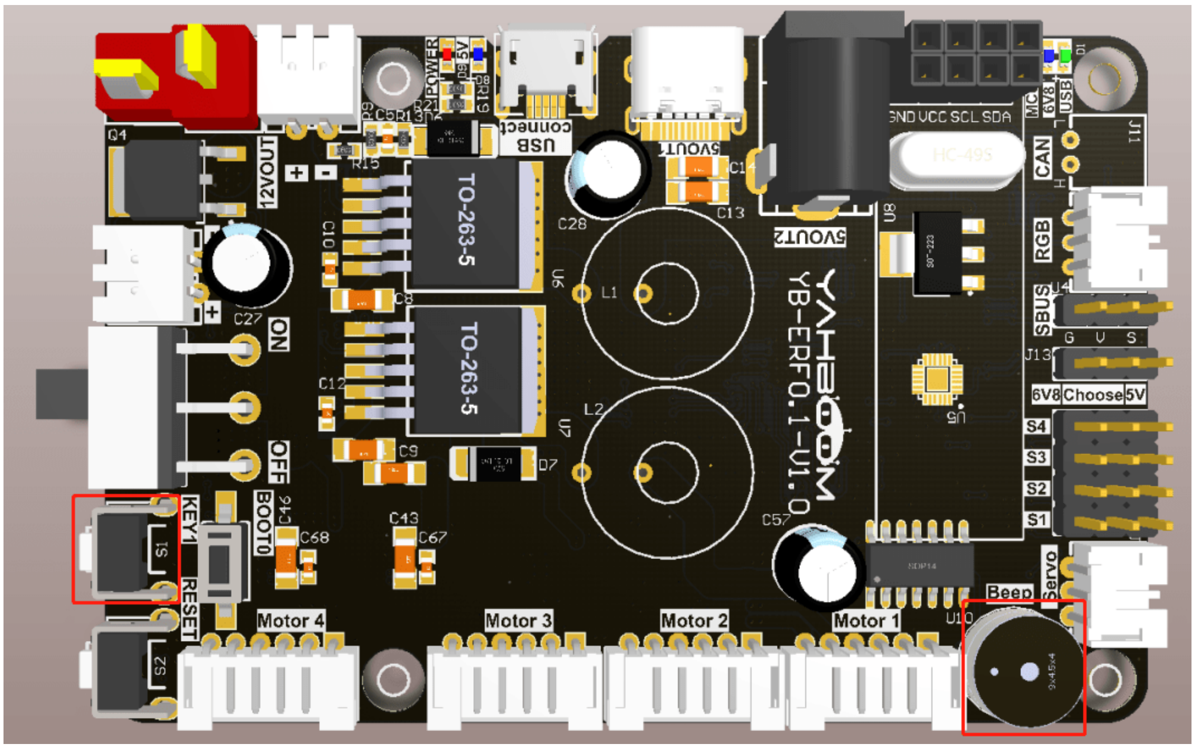
```
// The peripheral device is initialized  外设设备初始化
void Bsp_Init(void)
{
    Beep_On_Time(50);
}
```



```
// main.c中循环调用此函数，避免多次修改main.c文件。
// This function is called in a loop in main.c to
void Bsp_Loop(void)
{
    // Detect button down events   检测按键按下事件
    if (Key1_State(KEY_MODE_ONE_TIME))
    {
        Beep_On_Time(50);
    }

    Bsp_Led_Show_State_Handle();
    // The buzzer automatically shuts down when ti
    Beep_Timeout_Close_Handle();
    HAL_Delay(10);
}
```

3.6. Hardware Connection

The key KEY1 and buzzer are on-board components and do not need to be connected manually.



3.7. Experimental Effect

After burning the program, the buzzer will first sound for 50 milliseconds when powering on, the LED will flash every 200 milliseconds, and the buzzer will sound for 50 milliseconds every time a key is pressed.