南京理工大学

NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

计算机科学与工程学院

“嵌入式系统”实验报告书

题目：嵌入式系统作业 2——矩阵键盘扫描

学号：922106840127

姓名：刘宇翔

成绩

日期：　　2025 年　　3 月　　10 日

1 题目要求

1. 题目设计要求

【1】作业目标：

（1）熟悉嵌入式系统的寄存器工作原理；

（2）能够熟练掌握寄存器的 STM32 嵌入式程序开发；

（3）独自完成矩阵键盘的扫描程序功能。

【2】作业内容：

基于寄存器方法的完成教材 P121,图 6-4 的矩阵键盘的扫描程序开发。

【3】完成要求：

（1）基于寄存器库开发，不得使用 HAL 库；

（2）变成矩阵键盘的行列扫描功能，编制键码值生成程序，将按键 K1-K12，分别对应 1-0x0C 值，有按键按下一次时，调试串口输出响应的键码值；

（3）对按键具有软件消抖处理；

（4）程序代码附有必要的注释，并且编译通过无错，无须在开发板上测试。

2. 拟实现的具体功能

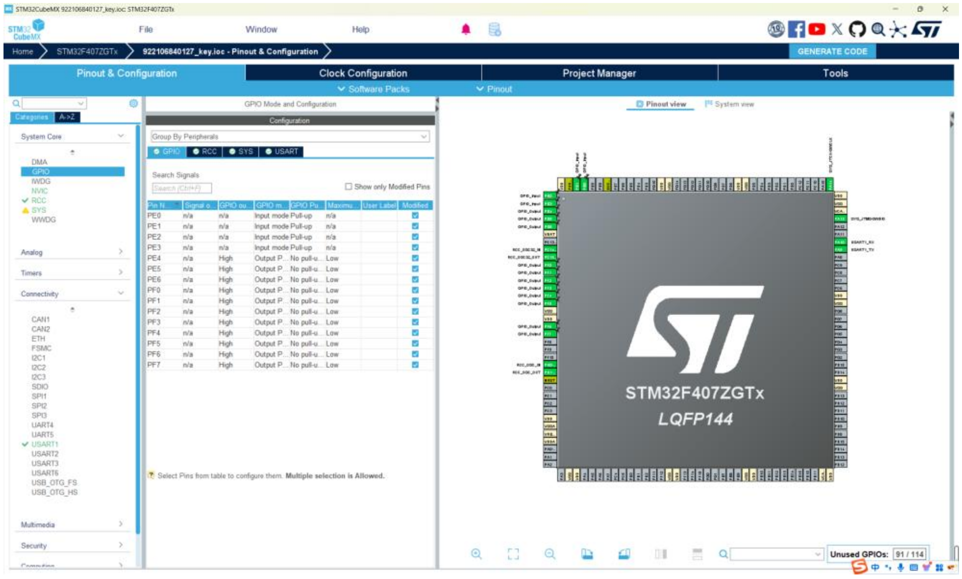
完成基于寄存器库开发的，能够实现矩阵键盘的行列扫描功能，编制键码值并生成程序，将对应的按键的键码值调整好，从而调试串口输出响应的键码值，并加入软件消抖处理，编译程序通过。

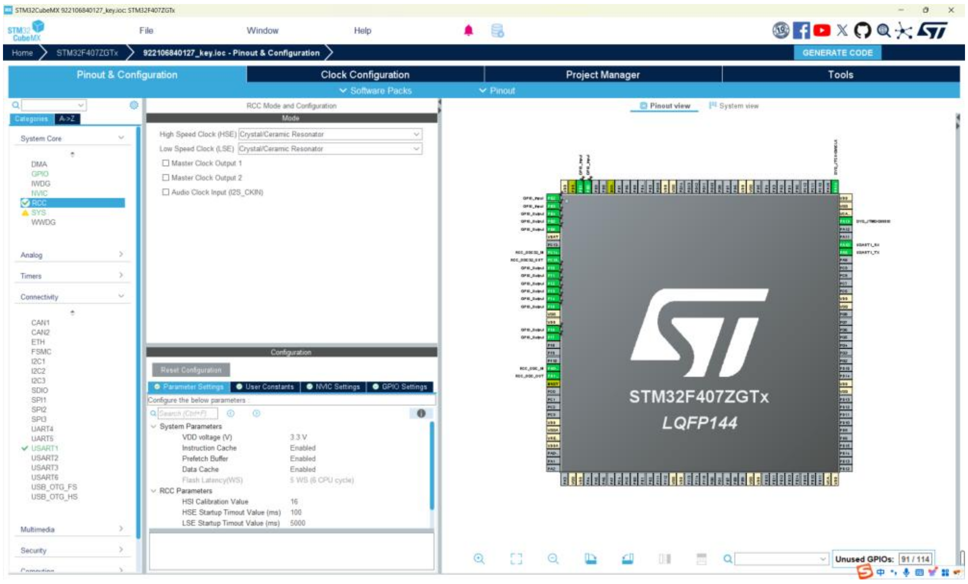
2 总体设计

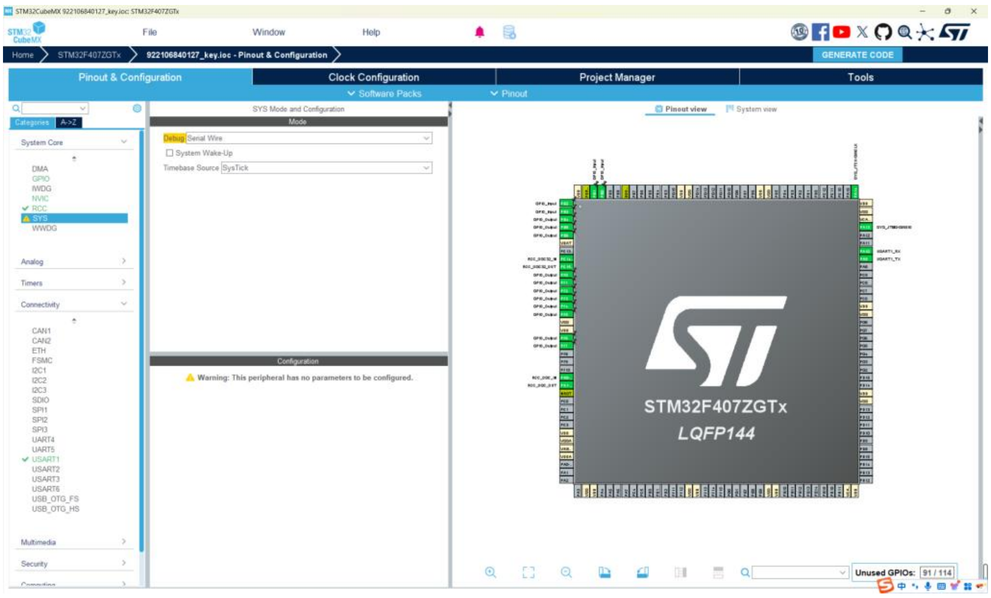
2.1 硬件设计

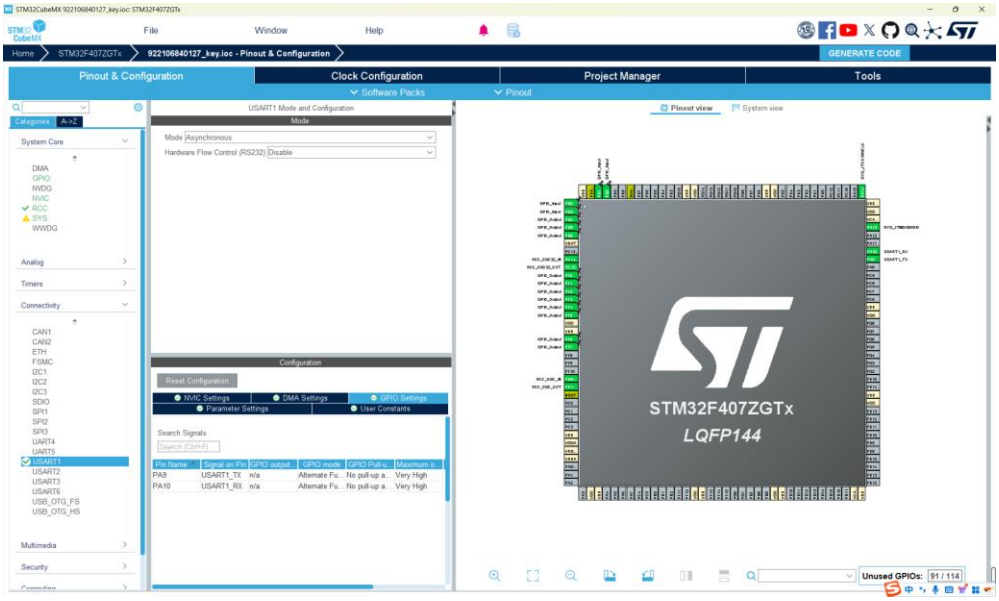
1. 硬件设计思路

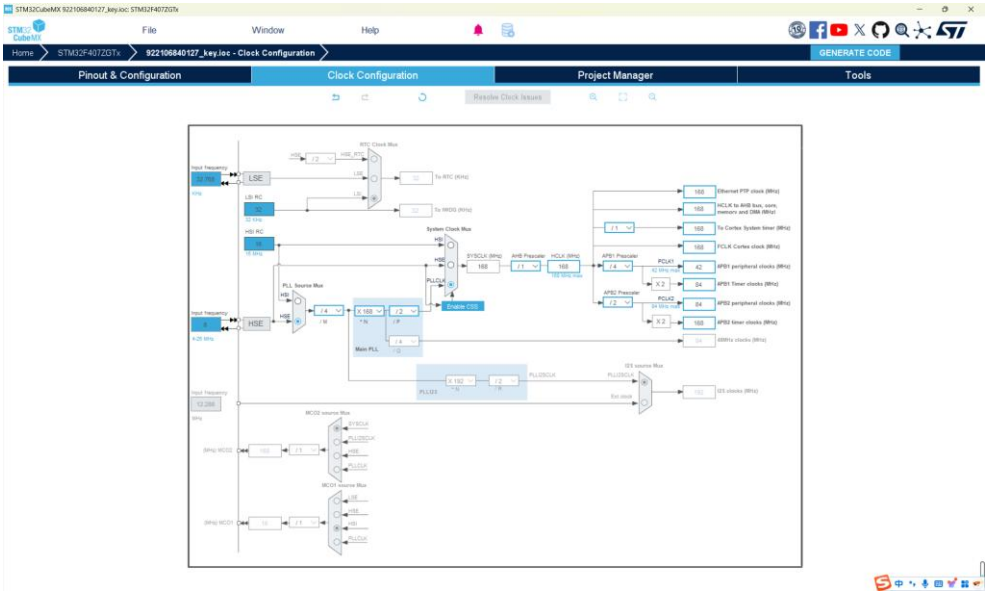
我通过查阅相关开发板原理图确定了在本次作业中会使用到的引脚并设置对应的属性值， 我使用 USART1 来实现串口调试的功能，具体代码工程配置如下：

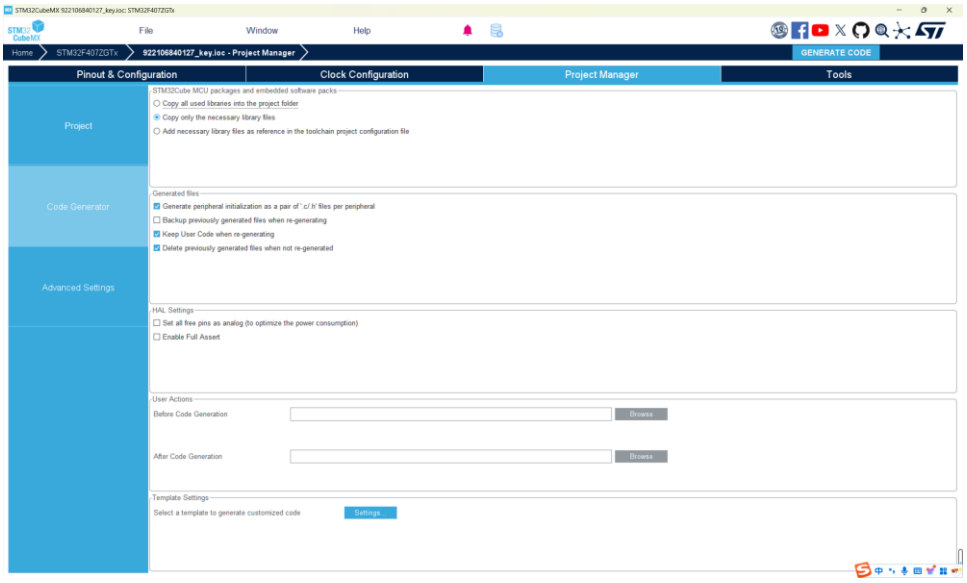

STM32CubeMX
File
Window
Help
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
GPIO Mode and Configuration
Configuration
Pinout view
System view
STM32F407ZGTx
LQFP144


STM32CubeMX
File
Window
Help
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
RCC Mode and Configuration
Mode
Configuration
Pinout view
System view
STM32F407ZGTx
LQFP144


STM32CubeMX
File
Window
Help
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
SYS Mode and Configuration
Mode
Configuration
Warning: This peripheral has no parameters to be configured.
Pinout view
System view
STM32F407ZGTx
LQFP144


File
Window
Help
Home
STM32F407ZGTx
922106840127_key.ioc - Pinout & Configuration
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
STM32F407ZGTx
LQFP144
Unused GPIOs: 91 / 114


File
Window
Help
Home
STM32F407ZGTx
922106840127_key.ioc - Clock Configuration
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
LSE
HSE
PLL Source Mux
System Clock Mux
Main PLL
Enable CSS


File
Window
Help
Home
STM32F407ZGTx
922106840127_key.ioc - Project Manager
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Project
Code Generator
Advanced Settings
STM32Cube MCU packages and embedded software packs
Copy all used libraries into the project folder
Copy only the necessary library files
Add necessary library files as reference in the toolchain project configuration file
Generated files
Generate peripheral initialization as a pair of '.c/.h' files per peripheral
Backup previously generated files when re-generating
Keep User Code when re-generating
Delete previously generated files when not re-generated
HAL Settings
Set all free pins as analog (to optimize the power consumption)
Enable Full Assert
User Actions
Before Code Generation
After Code Generation
Browse
Template Settings
Select a template to generate customized code
Settings...

以上配置图是我作为 STM32CUBEMX 进行的配置设置，设置后点击“Generate Code”生成初始化代码。

2.2 软件设计

1. 软件设计概述

本软件主要实现矩阵键盘扫描、软件消抖和调试串口输出功能。系统基于 STM32 单片机，采用直接操作寄存器的方法进行开发，不依赖高级库函数。软件设计主要包括以下几个模块：

（1） 系统初始化模块

包括系统时钟、GPIO 和 USART1 的初始化，确保各外设在预定频率下稳定运行。

（2） 矩阵键盘扫描模块

通过行列扫描方式检测矩阵键盘按键状态。利用预设映射数组将物理行转换为逻辑行，并根据检测到的列输入生成相应的键码值。

（3） 软件消抖模块

利用 SysTick 定时器实现微秒级延时，确保按键输入的稳定性，避免因机械抖动产生误触发。

（4） 串口调试输出模块

将生成的键码值转换为特定格式的字符串，并通过 USART1 以轮询方式逐字符发送，实现调试串口的数据输出。

总体而言，整体设计在硬件资源有限的条件下，通过直接寄存器操作实现了稳定可靠的按键检测和串口通信，有效提高了系统对外部输入信号的处理精度。

2. 软件流程图

（1）流程阶段分解

A. **初始阶段**

开始 → 系统初始化

程序启动后首先完成硬件初始化（如 GPIO 端口配置、USART 串口初始化等）。

B. **主循环结构**

进入主循环 → 遍历矩阵键盘每一行 (i = 0~2)

循环扫描矩阵键盘的 3 行（假设为 3xN 矩阵），通过逐行设置低电平检测列输入。

C. **按键检测流程**

设置当前行低电平 → 检测列输入状态

将当前扫描行置为低电平，读取列引脚的电平状态。

是否检测到按键？

如果**否**：直接跳转到**继续扫描下一行**

如果**是**：进入**消抖函数 → 再次检测列输入状态 → 确认按键有效？**

（双重消抖机制确保按键有效性）

D. **数据处理阶段**

根据列值计算键码 → KeyVal 是否有效？

将行列坐标转换为唯一键值，并校验其合法性。

如果**有效**：

构造包含键值的字符串 **→ 通过 USART1 发送字符串**

（如发送格式 "Key Pressed: X"）

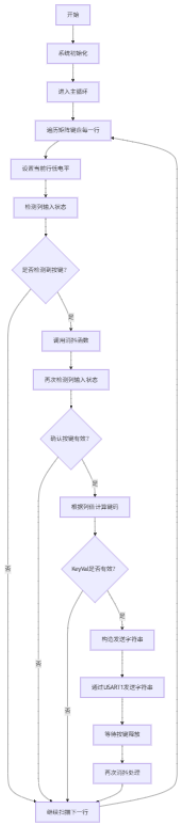
如果**无效**：跳过发送操作

E. **循环机制**

等待按键释放 → 再次消抖处理 → 继续扫描下一行

确保在用户松开按键前不会重复触发，完成当前行扫描后回到遍历循环起点。

（2）软件流程图展示



3. μ vision 详细代码

/* USER CODE BEGIN Header */

/**

****

****

* @file : main.c

* @brief : Main program body

* @details : 本程序实现矩阵键盘扫描及调试串口输出键码值功能，

* 包括软件消抖处理。本程序基于寄存器操作完成键盘扫描，

* 通过 USART1 调试串口输出按键对应的键码值。

****

****

```
  * @attention
  *
  * Copyright (c) 2025 STMicroelectronics.
  * All rights reserved.
  *
******************************************************************************
*********
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"         // 主头文件，包含全局配置及外设声明
#include "usart.h"        // USART 外设初始化及函数声明（基于寄存器初始化，但使用了 HAL 的函数）
#include "gpio.h"         // GPIO 外设初始化及函数声明

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
```

```
/* USER CODE BEGIN PM */
/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);   // 系统时钟配置函数
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/**
  * @brief   软件消抖函数
  * @param   nus: 消抖延时，单位：微秒
  * @details 该函数利用 SysTick 定时器当前值实现微秒级延时，用于对按键进行软件消抖，
  *                通过计算经过的计数值达到设定延时后退出函数。
  */
void eradicate_shake(uint32_t nus) {
    uint32_t told, tnow, tcnt = 0;      // told：上一次读取的 SysTick 值，tnow：当前 SysTick 值，tcnt：累计经过的计数值
    uint32_t ticks;                     // 需要延时的计数值
    ticks = nus * (SystemCoreClock / 1000000); // 根据系统时钟计算需要延时的总计数
    uint32_t reload = SysTick->LOAD + 1; // 获取 SysTick 的重载值（最大计数值
```

```
+1）
    told = SysTick->VAL;                    // 记录当前 SysTick 计数器的值

    // 循环直到累计计数达到所需延时
    while (1) {
        tnow = SysTick->VAL;                // 读取当前 SysTick 计数器的值
        if (tnow != told) {                 // 如果计数器发生变化
            // 如果当前计数值小于上一次值，则说明 SysTick 计数器正常递减
            if (tnow < told)
                tcnt += told - tnow;
            else
                // 如果当前计数值大于上一次值，说明 SysTick 已经溢出，从
reload 重新计数
                tcnt += reload - tnow + told;
            told = tnow;                    // 更新上一次计数值为当前值
            if (tcnt >= ticks)          // 当累计计数达到或超过设定值时退出循环
                break;
        }
    }
}
/* USER CODE END 0 */

/**
  * @brief  主程序入口函数
  * @retval int
  * @details 该函数完成初始化工作（HAL 初始化、系统时钟配置、GPIO 和
USART 初始化），
  *          进入主循环后不断扫描矩阵键盘，通过调试串口输出对应的键码
值。
  */
```

```
int main(void)
{
    uint16_t i, KeyVal = 0;   // i：用于行扫描的循环变量；KeyVal：存储计算后的按键键码值
    /* PreLine 数组用于将物理行映射为逻辑行：
       数组中的值用于按键值的计算，例如：
       如果 PreLine[i]=2，则该行对应的按键键码为 4*2 + (列号+1)
       即：当扫描到列 0 时，键码为 (4*2+1)=9，依此类推 */
    uint8_t PreLine[3] = {2, 0, 1};

    /* 用于存储调试串口发送字符串的数组，
       初始内容为 "KeyVal: "，后续追加按键值显示 */
    char txBuffer[16] = {'K','e','y','V','a','l',':',' '};
    /* 用于将键码值转换为 16 进制字符的查找表 */
    char hexChars[] = "0123456789ABCDEF";

    /* 初始化 HAL 库，复位外设，并配置 Flash 预取缓冲 */
    HAL_Init();
    /* 配置系统时钟，设置时钟频率、分频系数等 */
    SystemClock_Config();
    /* 初始化 GPIO 外设（矩阵键盘相关引脚） */
    MX_GPIO_Init();
    /* 初始化 USART1 外设（调试串口） */
    MX_USART1_UART_Init();

    /* 主循环：不断扫描矩阵键盘 */
    while (1)
    {
        /* 对矩阵键盘的行进行循环扫描，共 3 行 */
        for (i = 0; i < 3; i++) {
```

```
/* 激活当前行：
   通过设置 GPIOE 的输出数据寄存器（ODR），将当前扫描的行拉低，
   其余行保持高电平。
   假设行引脚位于 PE4、PE5、PE6，因此用 i+4 确定对应引脚 */
GPIOE->ODR = ~(1 << (i + 4));

/* 读取矩阵键盘的列输入：
   GPIOE 的低 4 位（PE0～PE3）对应矩阵键盘的 4 列，
   正常状态下由于上拉电阻应为高电平（0x0F），
   当有按键按下时对应位被拉低，不等于 0x0F */
if ((GPIOE->IDR & 0x0F) != 0x0F) {
    /* 消抖：调用 eradicate_shake 延时函数，延时 16,000 微秒（16ms） */
    eradicate_shake(16000);
    /* 再次判断按键状态，确认按键确实按下 */
    if ((GPIOE->IDR & 0x0F) != 0x0F) {
        /* 根据矩阵键盘检测到的列值生成对应的键码值
           根据 IDR 的低 4 位情况判断是哪一列按键按下，
           具体对应关系如下：
             0x0E (1110b) -> 第一列按键，键码 = 4 * PreLine[i] + 1
             0x0D (1101b) -> 第二列按键，键码 = 4 * PreLine[i] + 2
             0x0B (1011b) -> 第三列按键，键码 = 4 * PreLine[i] + 3
             0x07 (0111b) -> 第四列按键，键码 = 4 * PreLine[i] + 4 */
        switch (GPIOE->IDR & 0x0F) {
            case 0x0E:
```

```
                KeyVal = 4 * PreLine[i] + 1;
                break;
            case 0x0D:
                KeyVal = 4 * PreLine[i] + 2;
                break;
            case 0x0B:
                KeyVal = 4 * PreLine[i] + 3;
                break;
            case 0x07:
                KeyVal = 4 * PreLine[i] + 4;
                break;
            default:
                KeyVal = 0;   // 未识别的按键状态
                break;
        }
        /* 如果检测到有效的按键（KeyVal 不为 0） */
        if (KeyVal) {
            /* 构造用于串口发送的字符串
               格式为 "KeyVal: 0xXY\r\n"，其中 XY 为键码的
16 进制表示 */
            txBuffer[0] = 'K';
            txBuffer[1] = 'e';
            txBuffer[2] = 'y';
            txBuffer[3] = 'V';
            txBuffer[4] = 'a';
            txBuffer[5] = 'l';
            txBuffer[6] = ':';
            txBuffer[7] = ' ';
            txBuffer[8] = '0';
            txBuffer[9] = 'x';
```

```
                    /* 将键码高 4 位转换为对应 16 进制字符 */
                    txBuffer[10] = hexChars[(KeyVal >> 4) & 0x0F];
                    /* 将键码低 4 位转换为对应 16 进制字符 */
                    txBuffer[11] = hexChars[KeyVal & 0x0F];
                    txBuffer[12] = '\r';  // 回车符
                    txBuffer[13] = '\n';  // 换行符
                    txBuffer[14] = '\0';  // 字符串结束符

                    /* 通过 USART1 发送构造好的字符串 */
                    char *p = txBuffer;
                    while (*p) {
                        /* 等待 USART1 发送数据寄存器空（TXE 标志置位） */
                        while (!(USART1->SR & USART_SR_TXE));
                        /* 将当前字符写入数据寄存器，开始发送 */
                        USART1->DR = *p++;
                    }
                }
                /* 等待按键释放：
                   当所有列输入（PE0～PE3）均为高电平（0x0F）时，表示按键已释放 */
                while ((GPIOE->IDR & 0x0F) != 0x0F);
                /* 按键释放后再次进行消抖，防止按键抖动导致重复触发 */
                eradicate_shake(16000);
            }
        }
    }
}
```

```
/**
  * @brief System Clock Configuration
  * @retval None
  * @details 配置系统时钟，包括主振荡器、PLL、分频系数等，以确保系统按预期频率运行。
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};  // 振荡器初始化结构体
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};  // 时钟初始化结构体

  /** 配置主内部调节器电压输出
    * 通过 __HAL_RCC_PWR_CLK_ENABLE() 使能 PWR 时钟，
    * 然后配置电压调节等级为 SCALE1
    */
  __HAL_RCC_PWR_CLK_ENABLE();

__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** 初始化 RCC 振荡器，根据 RCC_OscInitStruct 配置参数
    * 配置 HSE（外部高速晶振）开启，
    * 启动 PLL，并将 PLL 源设为 HSE，
    * PLL 分频系数 PLLM = 4，倍频系数 PLLN = 168，
    * 系统时钟分频系数 PLLP = DIV2，PLLQ = 4
    */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
```

```
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 4;
  RCC_OscInitStruct.PLL.PLLN = 168;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 4;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    /* 如果振荡器配置失败，则调用错误处理函数 */
    Error_Handler();
  }

  /** 初始化 CPU、AHB 和 APB 总线时钟
    * 配置时钟源为 PLL 输出，同时设定 AHB、APB1、APB2 分频系数，
    * 以确保各总线工作在合适的频率下
    */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK |
RCC_CLOCKTYPE_SYSCLK
                              | RCC_CLOCKTYPE_PCLK1 |
RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) !=
HAL_OK)
  {
    /* 如果时钟配置失败，则调用错误处理函数 */
    Error_Handler();
  }
```

```
}

/* USER CODE BEGIN 4 */
/* USER CODE END 4 */

/**
  * @brief  当发生错误时调用的错误处理函数
  * @retval None
  * @details 该函数会禁止所有中断并进入无限循环，用于在系统错误时挂起程序。
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* 关闭所有中断 */
  __disable_irq();
  while (1)
  {
    /* 程序停在这里，等待复位 */
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  当断言失败时报告错误的源文件和行号
  * @param  file: 源文件名指针
  * @param  line: 出错行号
  * @retval None
  */
```

```
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

4. 函数设计与功能实现

（1） eradicate_shake（软件消抖函数）

功能概述

该函数利用 SysTick 定时器实现微秒级延时，用于对按键输入进行软件消抖处理，确保在按键抖动期间不会误判。

主要实现思路

计时原理：

根据传入的延时时间（单位微秒），利用系统核心时钟（SystemCoreClock）计算出需要延时的总计数值（ticks）。

利用 SysTick 计数器：

SysTick 是一个 24 位递减计数器，函数先获取当前计数值（told），然后在循环中不断读取当前值（tnow），并通过比较两次读取的差值来累计经过的计数（tcnt）。

处理计数器溢出：

当 SysTick 从零回到重载值时，会发生溢出，函数通过判断当前计数值与上一次计数值的关系来正确计算经过的计数。

退出条件：

当累计的计数值达到或超过所需延时的 ticks 后，函数退出，从而实现预期的延时效果。

（2）main（主函数）

功能概述

主函数完成系统初始化（包括 HAL、系统时钟、GPIO、USART1 等），进入无限循环后不断扫描矩阵键盘，并在检测到有效按键后，通过调试串口输出对应的

键码值。

主要实现流程

初始化阶段：

调用 HAL_Init() 完成 HAL 库的初始化工作。

调用 SystemClock_Config() 配置系统时钟。

分别调用 MX_GPIO_Init() 和 MX_USART1_UART_Init() 初始化 GPIO 和 USART1（这里初始化代码采用 HAL 库函数，但后续关键操作均直接操作寄存器）。

矩阵键盘扫描：

行扫描：

利用一个 for 循环对矩阵键盘的每一行进行扫描。程序通过设置 GPIOE 输出数据寄存器（ODR），将当前扫描行拉低，激活该行。

列读取：

通过读取 GPIOE 的低 4 位（假定 PE0～PE3 连接矩阵键盘的 4 列），检测是否有按键按下（正常状态下输入值应为 0x0F，由于内部上拉，当按键按下时对应位被拉低）。

消抖处理：

在检测到按键按下后，先调用 eradicate_shake(16000) 进行约 16ms 的延时，再次判断按键状态以确保稳定。

键码值生成及串口发送：

根据不同列的检测值（如 0x0E、0x0D、0x0B、0x07），通过 switch-case 结构计算出按键对应的键码值。计算公式为：

KeyVal = 4 * PreLine[i] + (列序号 + 1)

其中 PreLine 数组用于将物理行映射为逻辑行。

构造输出字符串：将 "KeyVal: 0x" 与键码值的 16 进制表示组合成一个字符串，例如 "KeyVal: 0x05\r\n"。

通过 USART1 输出：

利用 USART1 的状态寄存器（SR）检测 TXE 标志位，逐字符将字符串写入数据寄存器（DR）发送出去。

按键释放检测：

发送完成后，程序等待直到所有列输入恢复为高电平（表示按键已释放），并再次调用消抖函数以防止由于按键抖动而重复触发。

（3）SystemClock_Config（系统时钟配置函数）

功能概述

该函数利用 HAL 库的配置接口来初始化系统时钟，确保微控制器在预期的频率下运行。

主要实现流程

电压调节与时钟源使能：

首先通过使能 PWR 时钟和配置电压调节等级（PWR_REGULATOR_VOLTAGE_SCALE1）确保系统电源稳定。

外部晶振与 PLL 配置：

启用 HSE（外部高速晶振）。

配置 PLL，使其以 HSE 为输入，设置分频系数（PLLM）、倍频系数（PLLN）、以及 PLLP、PLLQ 参数，确保系统时钟达到所需频率。

总线时钟配置：

将系统时钟源设置为 PLL 输出，同时配置 AHB、APB1 和 APB2 总线的分频系数，保证各总线及外设获得合适的时钟频率。

错误处理：

如果任一配置步骤返回错误，调用 Error_Handler() 进行错误处理。

（4）Error_Handler（错误处理函数）

功能概述

当系统遇到配置错误或其他严重故障时，该函数会被调用，用于停止系统继续运行并进入安全状态，便于调试和错误诊断。

主要实现流程

中断关闭：

调用 __disable_irq() 关闭所有中断，防止错误状态下出现意外操作。

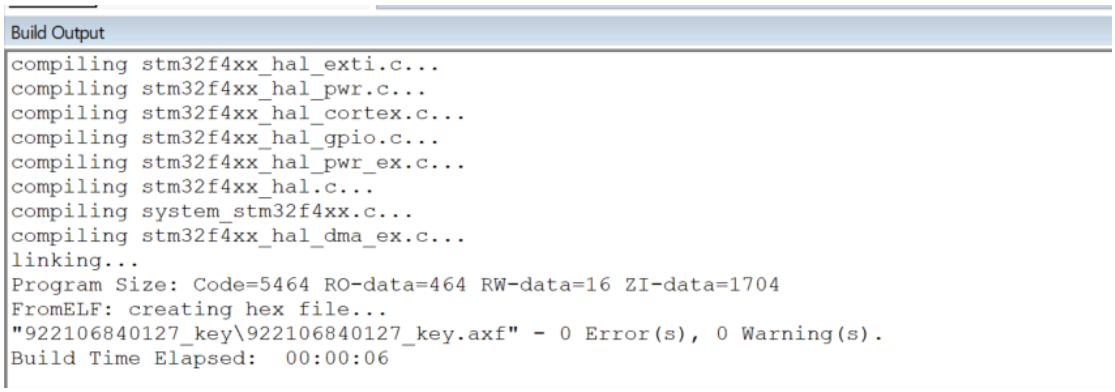
无限循环：

程序进入一个无限循环，确保系统停在此处，便于开发人员调试或等待复位。

可扩展性：

在实际项目中，用户可以在该函数中加入错误记录、LED 指示或其他错误上报机制，以便更好地定位问题。

3 实验结果分析与总结



编译通过，无报错，目标题设功能已实现，完成题目要求。