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：ex2_922106840127_Uart

学号：922106840127

姓名：刘宇翔

成绩

日期：　　2025 年　　4 月　　7 日

1 题目要求

1. 题目设计要求

（1）作业内容：

用串口 1 来进行异步串行通信，发送数据时采用状态查询方式进行发送，接收数据时(长度不超过 200 字节)采用中断方式进行接收，当收到回车、换行符时，将收到的字符串通过串口返回给 PC。

（2）完成要求：

工程名称命名：ex2_学号_Uart，并打包成：ex2_学号_ Uart.rar 压缩文件夹

实验报告 PDF 格式：ex2_学号_ Uart.pdf

2. 拟实现的具体功能

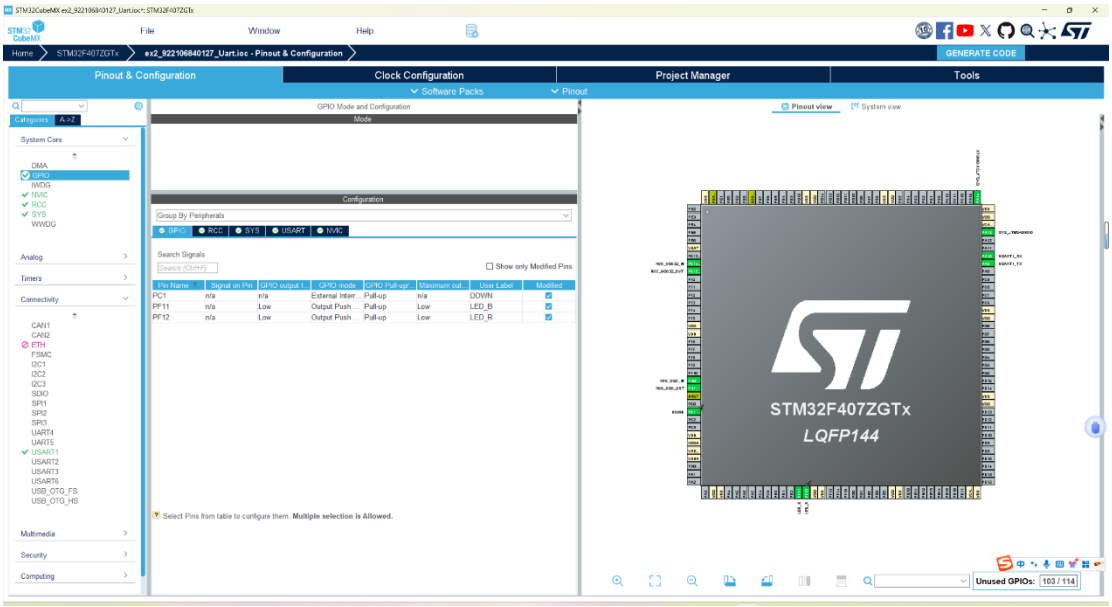
本实验基于 STM32 单片机和 HAL 库，通过 UART 中断实现串口数据实时接收与回显功能。系统在初始化阶段完成了时钟、GPIO 和 USART 外设配置，并启动中断接收模式。实验过程中，单片机每次接收到一个字符后，将其存储于接收缓冲区，并检测该字符是否为换行符；一旦检测到换行符，系统立即在缓冲区末尾添加字符串结束符，并利用 UART 回传整行数据，实现数据的原样回显。该设计充分利用中断机制保证数据处理的实时性，并通过缓冲区管理提高通信的完整性和可靠性，为后续实现更复杂的数据交互提供了坚实基础。

2 总体设计

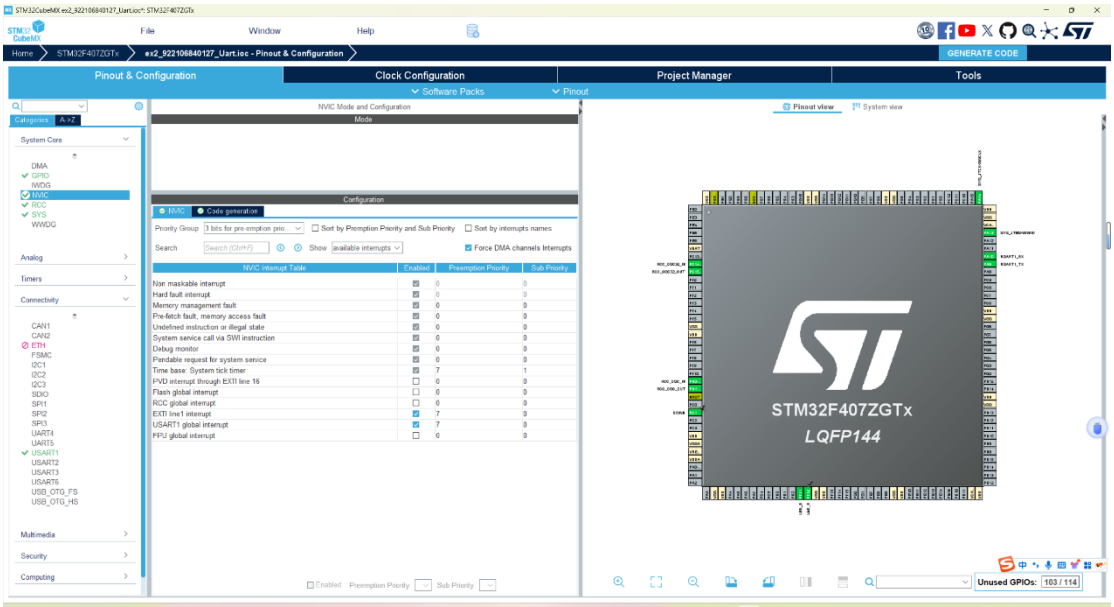
2.1 硬件设计

1. 硬件设计思路

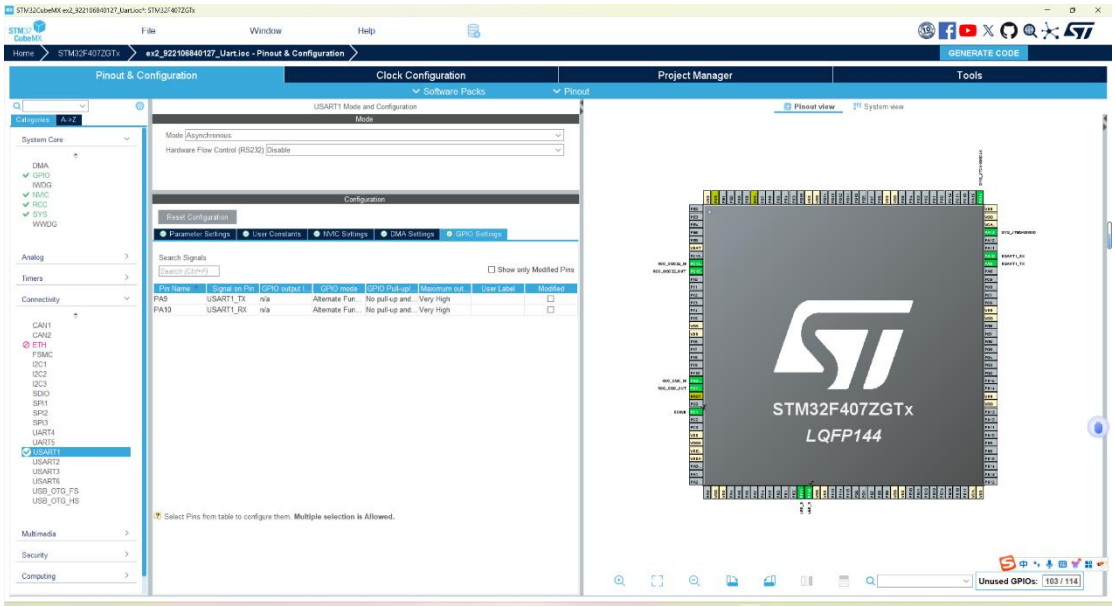
我通过查询相关开发板原理图确定了本次实验所需要的有关 GPIO 端口、NVIC 设置以及 USART 通信端口的设置，具体的代码工程配置硬件设计如下：



GPIO 设置



NVIC 设置



USART 设置

如图所示，我在 NVIC 的相关设置界面对几种中断的相关属性做了具体的规定，USART 的通信模式采用的是异步通信。

其他设置与前几次实验相同，如 RCC 采用 Crystal/Ceramic Resonator，使用 Serial Wire 用于调试对应的 debug 接口，对 Clock Configuration 的时钟配置进行规范，选择了 MDK-ARM V5.32 作为编译工具链，其他内容此处省略。

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

本软件主要实现基于串口中断的字符接收与回显功能，通过 STM32 单片机接收上位机发送的字符数据，并在接收到换行符时将整行数据通过串口回传，实现简单的串口通信功能。系统基于 HAL 库进行外设初始化，并采用中断方式实现对串口接收的实时响应，整体设计结构简洁，模块功能清晰，具有良好的稳定性与可扩展性。具体设计内容如下：

（1） **系统初始化模块**

在 main 函数中，系统首先通过 HAL_Init() 完成底层硬件的复位与初始化，随后调用 SystemClock_Config() 配置系统时钟以确保各模块稳定运行。接着依次

调用 MX_GPIO_Init() 与 MX_USART1_UART_Init() 对 GPIO 与串口外设进行初始化。初始化完成后，通过 HAL_UART_Receive_IT() 启动串口中断接收功能，使系统能够实时接收上位机发送的数据。

接着，我对我在本实验中要用到的一些变量进行了相关的设置：

```
#define MAX_RX_BUFFER_SIZE 200   // 接收缓冲区大小
uint8_t rx_buffer[MAX_RX_BUFFER_SIZE];   // 接收缓冲区
uint8_t rx_index = 0;                    // 当前接收位置
```

依据题意要求设置接收缓冲区为 200，设置对应的数组变量作为接收缓冲区，并添加一个索引变量用于指示当前接受字符串到的缓冲区位置。

（2） 串口中断处理模块

系统通过串口接收中断 (HAL_UART_RxCpltCallback) 实现对每个接收到字符的处理逻辑。每次接收到一个字符后，判断是否为换行符：

- 若为换行符（\r 或 \n），表示一行数据接收完成，调用 ProcessReceivedData() 函数对接收到的数据进行回显，并添加换行，同时清空接收缓冲区索引；
- 若为其他字符，则继续将字符存入接收缓冲区，等待下一次中断。

为提高代码可读性与结构清晰度，我专门设计了 IsNewLineChar() 函数用于判断函数用于判断是否为换行字符。

（3） 数据处理模块

ProcessReceivedData() 函数用于处理完整接收到的一行字符串。函数将接收缓冲区末尾添加字符串结束符 \0，随后调用 HAL_UART_Transmit() 发送回上位机，实现数据回显功能，并附加换行，最后重置接收索引 rx_index 为 0，准备接收下一行数据。

（4） 换行符判断模块

设计 IsNewLineChar(uint8_t ch) 函数对输入字符是否为换行符进行判断，支持识别 \r 和 \n，从而增强代码的可读性和模块化程度，使得判断逻辑更加直观、便于维护和扩展。

总体而言，本软件在 STM32 平台下，利用中断机制与模块化设计，实现了

对串口输入数据的实时接收与回显，结构清晰，运行稳定，为后续扩展串口通信相关功能打下了良好基础。

3. 软件流程分解

A. 初始阶段

开始 → 系统初始化

程序启动后，首先完成硬件与外设初始化，包括：

- 系统底层初始化（调用 HAL_Init() 对 MCU 进行复位与初始化）
- 系统时钟配置（通过 SystemClock_Config() 设置 PLL 及各时钟分频参数，确保 MCU 与外设稳定运行）
- GPIO 与串口外设初始化（分别调用 MX_GPIO_Init() 与 MX_USART1_UART_Init() 为后续数据传输提供硬件支持）
- 启动 UART 中断接收（使用 HAL_UART_Receive_IT() 启动接收中断，实现对字符数据的实时接收）

B. 主循环结构

进入主循环 → 系统处于等待状态

在 main 函数中，经过初始化后系统进入主循环，此时主要任务为等待串口接收中断触发，并无其他循环任务。

C. UART 接收中断触发流程

数据传输 →UART 中断产生

当上位机发送数据时，每接收到一个字节均会触发 UART 中断，系统自动进入中断回调函数 HAL_UART_RxCpltCallback()，处理当前接收到的字符。

D. 数据接收与换行判断流程

进入中断回调函数 → 判断接收字符

在 HAL_UART_RxCpltCallback() 内，判断当前接收到的字符是否为换行符：

- 调用 IsNewLineChar() 函数检测当前字符是否为 '\r' 或 '\n'
- 若判断为换行符，则认为一行数据接收完毕，进入数据处理阶段
- 若非换行符，则递增接收缓冲区索引，等待下一字符

E. 数据处理与回显流程

检测到换行符 → 调用数据处理函数

在 ProcessReceivedData() 中，完成以下操作：

- 在接收缓冲区末尾添加字符串结束符（\0），形成完整的字符串
- 通过 HAL_UART_Transmit() 将接收到的字符串原样回传，并附加换行符，实现数据回显
- 重置接收缓冲区索引，为下一次数据接收做准备

F. 循环机制

数据处理完毕 → 重新启动接收中断 → 返回主循环

在每次数据处理后，系统重新调用 HAL_UART_Receive_IT() 启动下一字节的接收，完成一次完整的中断服务流程后，系统返回主循环，持续等待新的串口数据接收。

4. μvision 详细代码

```
#include "main.h"
#include "usart.h"
#include "gpio.h"
#include <string.h>

#define MAX_RX_BUFFER_SIZE 200   // 接收缓冲区大小

uint8_t rx_buffer[MAX_RX_BUFFER_SIZE];   // 接收缓冲区
uint8_t rx_index = 0;                    // 当前接收位置

void SystemClock_Config(void);
uint8_t IsNewLineChar(uint8_t ch);
void ProcessReceivedData(void);

/**
  * @brief 判断是否为换行符
  * @param ch: 字符
  * @retval 1 是换行符，0 否
  */
```

```
uint8_t IsNewLineChar(uint8_t ch)
{
    return (ch == '\r' || ch == '\n');
}

/**
  * @brief 处理接收到的一行数据
  */
void ProcessReceivedData(void)
{
    rx_buffer[rx_index] = '\0';   // 添加结束符
    HAL_UART_Transmit(&huart1, rx_buffer, rx_index, HAL_MAX_DELAY);   // 回显
    HAL_UART_Transmit(&huart1, (uint8_t *)"\r\n", 2, HAL_MAX_DELAY); // 添加换行
    rx_index = 0;   // 重置索引
}

/**
  * @brief 串口接收中断回调
  * @param huart: 串口句柄
  */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart == &huart1)
    {
        if (rx_index < MAX_RX_BUFFER_SIZE - 1)
        {
            if (IsNewLineChar(rx_buffer[rx_index]))
            {
```

```
                ProcessReceivedData();  // 检测到换行符，处理数据
            }
            else
            {
                rx_index++;  // 等待下一字符
            }
        }
        else
        {
            rx_index = 0;  // 溢出保护，清空
        }

        // 启动下一次接收
        if (HAL_UART_Receive_IT(&huart1, &rx_buffer[rx_index], 1) !=
HAL_OK)
        {
            Error_Handler();
        }
    }
}

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    // 启动串口接收中断
    if (HAL_UART_Receive_IT(&huart1, &rx_buffer[rx_index], 1) != HAL_OK)
```

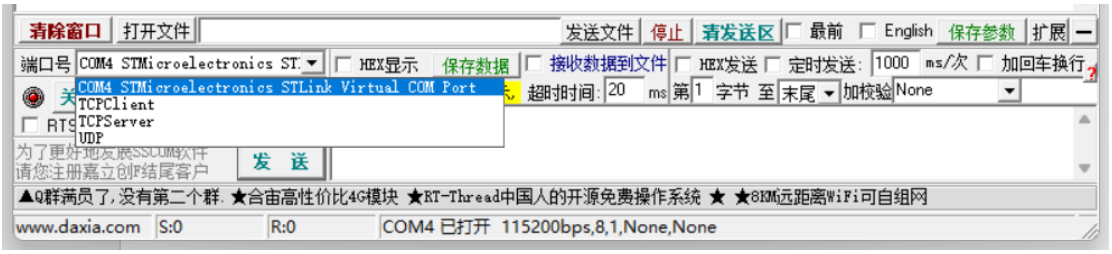
```
    {
        Error_Handler();
    }
    while (1){}
}

void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    __HAL_RCC_PWR_CLK_ENABLE();

__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_S
CALE1);

    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)
    {
        Error_Handler();
    }
```

```
    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)
    {
        Error_Handler();
    }
}

void Error_Handler(void)
{
    __disable_irq();
    while (1){}
}

#ifdef USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{

}
#endif /* USE_FULL_ASSERT */
```

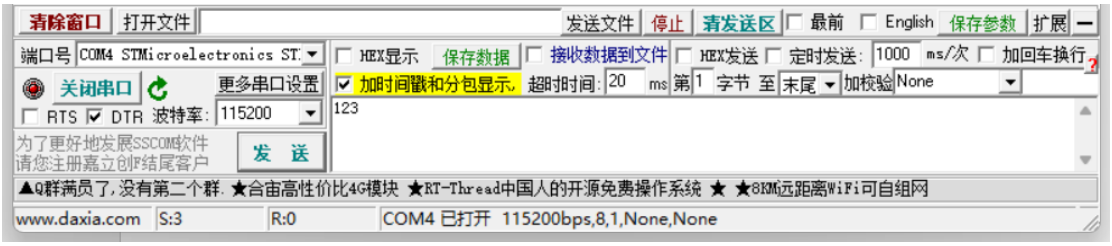
3 实验结果分析与总结

```
Build Output
compiling main.c...
linking...
Program Size: Code=5164 RO-data=448 RW-data=20 ZI-data=1300
FromELF: creating hex file...
".\ex2_922106840127_Uart\ex2_922106840127_Uart.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

对项目文件进行整体编译后上板实现，使用 SSCOM V15.3.1 软件用于监听 UART 串口通信，一些 SSCOM 的具体设置如下：



端口号采用 STLink 的端口，打开串口调试。



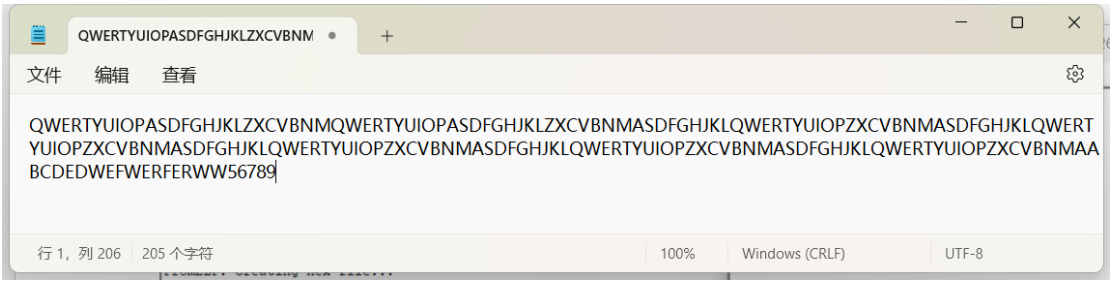
使用 DTR 模式，波特率设置为 115200，与前文所述的硬件设计实现一致。

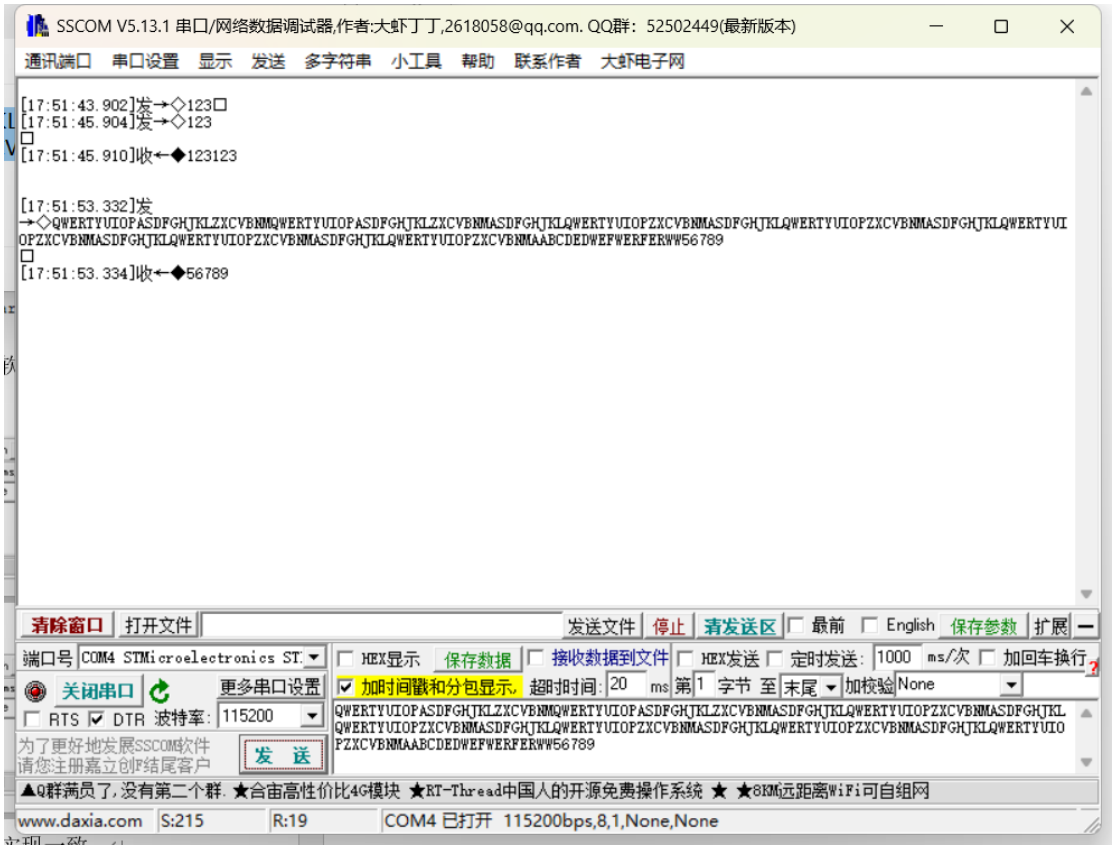
接下来，进行串口通信测试：

首先，我先发送一个不带换行符的“123”，然后检查 SSCOM 是否有返回字符串，检验不带换行符不返回缓冲区字符串的题目要求；

其次，我再发送一个末尾带换行符的字符串“123”检查程序是否在检测到换行符后将字符串缓冲区的所有内容发送回来；

最后，我设计了一个长度为 205 的字符串，用于检验发送字符串大于 200 时的函数功能设计实现是否符合预期，在第 201 个字符位置设置为数字，其余位置使用不同的大小写英文字母填充，查看最终效果。





从上文实验结果图可以看出，发送 123 不带换行符时开发板不会立即返回字符串 123，而是暂存在字符串缓冲区中，发送 123 带换行符的时候开发板会将这个新的 123 与缓冲区已有的 123 一起返回显示，原缓冲区清零。

发送字符串中低于 200 的位置由于超出缓冲区而全部丢弃，返回的是 201-205 个字符组成的字符串：56789，完成实验要求与目的。