

串口实验报告

吴立文 23061925

1. 实验要求

针对按键扫描和数码管实验，撰写实验报告，至少包含以下内容：

- 1) 通过原理图，对照实验板，确认串口连接的引脚、串口号、物理接口等信息；
- 2) 附件的 `uart.c` 加入到先前的工程，阅读程序，在 `main.c` 中完成对串口的配置和相关函数调用，通过 `sscom` 串口助手尝试给板子发送信息，在中断服务程序读取串口接收寄存器的地方打断点，验证串口数据接收情况；请截图记录进入断点时的串口相关寄存器值。
- 3) 板子 CPU 给 PC 发送一个字节数据，查看 PC 串口数据收发情况，验证 CPU 发送串口数据功能；
- 4) 参照给的示例，确认数据接收完整一帧的情况（连续多个字节），编写程序，完成发送一帧数据（通过中断服务中发送多个字节实现，注意 TC 和 TXE 的区别。通过 `watch` 窗口查看数据收发的情况；
- 5) 实现 Modbus RTU 协议 0x03 和 0x10 功能码；
- 6) 串口帧的完整性判断。

通常以字符之间的时间间隔作为帧的结束标志。例如串口收到一串数据后，如果超过 3.5 个字符（Modbus 标准，实际实现可以自由调整，例如 10ms）传输时间没有收到新的数据，则认为一帧结束。

2. 实验设计

2.1. USART 通信实现

本实验在 STM32F103 平台上，通过对 GPIO、USART 和 NVIC 的配置，基于串口在接收数据（RXNE）以及发送完成（TC）时自动触发的中断，实现了 USART 通信。下面首先对中断的初始化过程进行简要说明。

首先，需要对串口相关的 GPIO 口进行复用功能配置。USART1 使用 PA9 作为发送端（TX），PA10 作为接收端（RX）。因此在初始化中，将 PA9 设置为复用推挽输出模式（AF_PP），用于输出串口数据；将 PA10 设

置为浮空输入模式，用于接收外部设备发送的数据。GPIO 配置完成之后，再使能对应的 USART1 时钟，为串口模块工作提供基础条件。

随后，对 USART1 进行初始化，设置包括波特率、数据位、停止位、校验位以及硬件流控制等参数。本实验采用 115200 波特率、8 数据位、1 停止位、CRC16 校验和无流控。完成参数设置后，通过 USART_Init() 函数写入到串口寄存器中，并启用串口模块。

为了让串口具备中断响应能力，需要调用 USART_ITConfig() 打开对应的中断源。本实验中启用了两个中断：

- USART_IT_RXNE：当串口接收缓冲区非空时触发，用于接收外部设备发送的数据。
- USART_IT_TC：当一个字节发送完成后触发，用于在中断中继续自动发送后续数据，实现异步发送流程。

启用中断后，还需要在 NVIC 中配置中断优先级。NVIC 负责管理整个系统的中断响应，因此需为 USART1 指定抢占优先级和子优先级。本实验选择抢占优先级 2、子优先级 0，并在 NVIC 初始化结构体中启用 USART1_IRQn 通道。配置完成后，中断控制器即可在串口事件发生时响应中断请求。

通过以上步骤，系统便能够在收到数据（RXNE）时主动进入接收中断，在每发送完一个字节（TC）后进入发送完成中断，从而实现基于中断的 UART 通信流程。核心中断处理函数代码如下：

```
void USART1_IRQHandler(void) {  
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) { //接收中断  
        RecieceFlag = 1;  
        FrameFlag = 0;  
        if(RXPos<=USART_BUF_LEN) { // 缓冲区未满  
            USART_Rxbuf[RXPos]=USART_ReceiveData(USART1); // 读取接收到的数据 ;  
            RXPos++;  
        }  
        RecvTimeOver = 10; //每次接收到数据，超时检测时间 10ms  
    }  
  
    if (USART_GetITStatus(USART1, USART_IT_TC) != RESET) { // 发送中断  
        USART_ClearITPendingBit(USART1, USART_IT_TC);  
        if (SendPos < SendBufLen){  
            USART_SendData(USART1,USART_Txbuf[SendPos]);  
        }  
    }  
}
```

```
        SendPos++;  
    }  
}  
}
```

配置好了中断，实现 USART 通信就较为方便了：每次触发中断时，我们检查中断来源，需要接受就把数据写入缓冲区，需要发送就把数据发送出去，即完成了 USART 通信功能的实现。代码中的 `FrameFlag`、`ReceiveFlag`、`RecvTimeOver` 变量的作用后续会做说明。

2.2. 数据帧切分

Modbus RTU 协议下规定 3.5 个字符通信时间下没有传输发生，就算当前帧完成了发送，在本次实验中不妨取这个时间间隔为 10ms。

先前数码管扫描显示实验中，我们实现了一个计时器中断。那个实验中计时器中断是根据我们需要的扫描频率设置触发时间，然后取触发数码管位选切换。我们稍稍改造一下这个计时器中断：我们把中断改为固定 1ms 触发一次，所有与计时有关的逻辑都根据这个中断来计时，修改后代码如下：

```
void SysTick_Handler(void)  
{  
    digit_switch_cnt--;  
    if (digit_switch_cnt <= 0)  
    {  
        digit_switch_cnt = digit_switch_cnt_const;  
        digit_switch_flag = true;  
    }  
  
    RecvTimeOver--;  
    if (RecvTimeOver <= 0){  
        RecvTimeOver = 0;  
        FrameFlag = 1;  
    }  
}
```

`digit_switch_cnt` 是给数码管位选切换用的计数变量，根据所需要的时间设定 `digit_switch_cnt_const`，便能完成循环计时，前台扫描 `digit_switch_flag` 便能知道是否需要切换位选。

`RecvTimeOver` 也是同理，当收到数据触发 USART 中断我们会把它的值设为 10 并把 `FrameFlag` 置 0，当 `RecvTimeOver` 递减至 0 时再将 `FrameFlag` 置

1, 结合 ReceiveFlag, 当 ReceiveFlag 与 FrameFlag 同时为 1 时, 说明我们收到了数据并且完成了一帧数据的接收。此时 USART_Rxbuf 中便保存了接收到的这一帧数据。

2.3. CRC 校验

CRC 校验按照特定规则, 将数据序列进行多项式运算, 得到一个用于检验数据完整性的校验值。其基本思想是把整个数据看成一个二进制多项式, 通过与约定好的生成多项式相除, 最终得到的余数作为 CRC 校验码。当接收端收到数据后, 再按照相同的多项式规则进行计算, 如果得出的余数与报文中的 CRC 一致, 则认为数据可靠; 若不一致, 则说明数据在传输中出现了错误。

CRC 的实际运算方式包括移位、异或等步骤, 流程虽然固定, 但具体细节会因标准不同而变化, 例如 Modbus 协议使用 CRC-16 (多项式 0xA001)。在程序实现时, 一般不需要手动推导全部运算步骤, 只需参考协议或标准中给出的多项式和初始值, 即可根据这些参数编写对应的计算函数或查表进行校验。

具体实现时, 根据 CRC-16 标准给出的初值与多项式, 完成相应的 CRC 计算即可。

2.4. 解析并执行 Modbus 协议

Modbus 协议的大致结构如下 (不同功能码略有区别):

主机（PC）请求							
设备地址（实验板子，自行指定）	功能码	第一个寄存器的高位地址	第一个寄存器的低位地址	寄存器的数量的高位	寄存器的数量的低位	CRC错误校验	
01	03	00	08	00	01	04	7F
从机（板子）应答							
地址	功能码	字节数	数据高字节	数据低字节	CRC错误校验		
01	03	02	41	24	88 0f		
十六进制数4124表示的十进制整数为16676。							

本次实验中共计划实现了 0x0000~0x0003 四个寄存器, 分别对应 4 个数码管显示的内容 (digit 数组)。设备地址固定为 0x01, 通过实现 0x03 与 0x10 两个功能码实现对数码管显示内容的读与写。并支持异常码返回。

根据先前的实现, 现在我们已经将一帧数据保存在了 USART_Rxbuf 数组中。首先我们要检验设备地址与 CRC 校验码, 如果匹配不上直接根据异常码规则返回错误信息。

2.4.1. 0x03（读）

核心代码如下：

```
if (USART_Rxbuf[1] == 0x03) { // 处理读
    reg_addr = (USART_Rxbuf[2] << 8) + USART_Rxbuf[3];
    reg_num = (USART_Rxbuf[4] << 8) + USART_Rxbuf[5];

    if (reg_addr + reg_num - 1 >= 4) { // 读越界
        Err(0x03, 0x02);
        return 0;
    }

    SendBufLen = 0;
    USART_Txbuf[SendBufLen++] = DeviceID;
    USART_Txbuf[SendBufLen++] = 0x03;
    USART_Txbuf[SendBufLen++] = 0x02 * reg_num; // 字节数

    for (i = reg_addr; i < reg_addr + reg_num; i++) {
        temp = digit[i] & 0xFFFF;
        USART_Txbuf[SendBufLen++] = temp >> 8;
        USART_Txbuf[SendBufLen++] = temp & 0xFF;
    }

    Send();
    return 1;
}
```

Err(功能码, 异常码)函数能自动生成异常码数据帧, 并调用 **Send()**函数完成发送。**Send()**函数在发送前会为发送缓冲区（**USART_Txbuf**）数据添加 **CRC** 校验码。

我们根据协议, 解析出寄存器地址与寄存器数数量后, 访问 **digit** 数组中对应内容, 写入发送缓冲区即可, 最后调用发送即可。

中途如果出现越界, 调用 **Err** 函数并传入相应信息即可。

2.4.2. 0x10（写）

核心代码如下：

```
if (USART_Rxbuf[1] == 0x10) { // 处理写
    reg_addr = (USART_Rxbuf[2] << 8) + USART_Rxbuf[3];
    reg_num = (USART_Rxbuf[4] << 8) + USART_Rxbuf[5];
```

```
data_len = USART_Rxbuf[6];

if (reg_addr + reg_num - 1 >= 4){
    Err(0x10, 0x02);
    return 0;
}

if (data_len != reg_num * 2){
    Err(0x10, 0x03);
    return 0;
}

for (i = 0; i < reg_num; i++) {
    temp = (USART_Rxbuf[7 + (2 * i)] << 8) + (USART_Rxbuf[7 + (2 * i + 1)]);
    if (temp > 15) {
        Err(0x10, 0x03);
        return 0;
    }
    digit[reg_addr + i] = temp;
}

SendBufLen = 6;
USART_Txbuf[0] = DeviceID;
USART_Txbuf[1] = 0x10;
USART_Txbuf[2] = USART_Rxbuf[2];
USART_Txbuf[3] = USART_Rxbuf[3];
USART_Txbuf[4] = USART_Rxbuf[4];
USART_Txbuf[5] = USART_Rxbuf[5];

Send();
return 1;
}
```

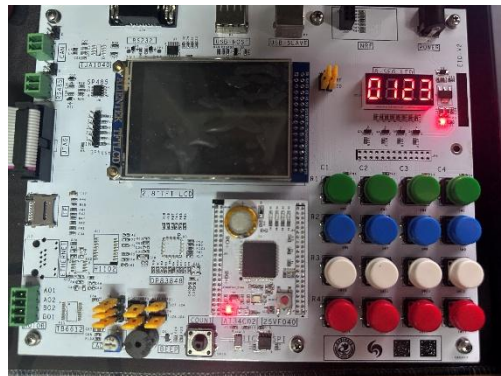
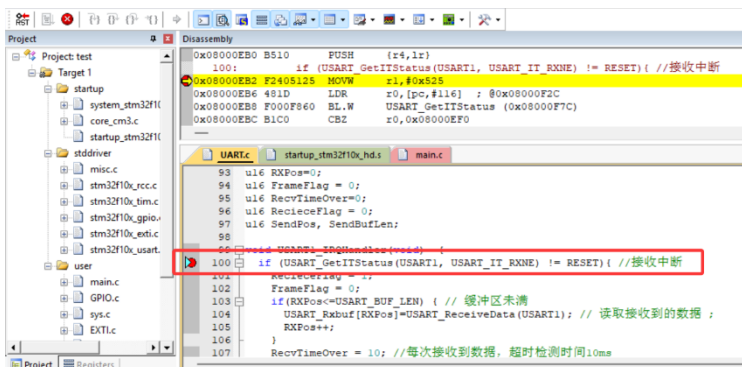
写寄存器的实现完全类似读寄存器，只需要额外注意写入的字节数要与待写的寄存器数相同，且写入的数据要能合法显示（0~15）。完成写入后按照协议要求发送相应信息即可。

在完成一帧数据的处理后，还需要记得把 **RecieceFlag** 与 **FrameFlag** 重新置为 0，等待下一帧数据输入。

3. 实验结果

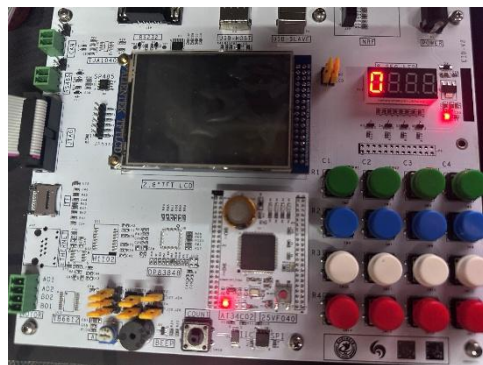
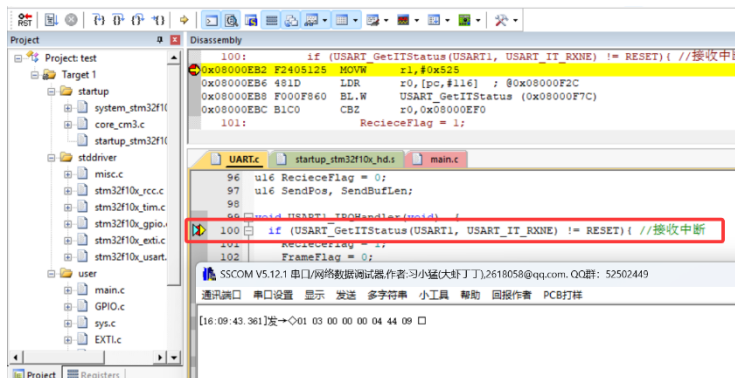
3.1. 检验 USART 中断

我们 USART 中断函数首行打上断点，然后进入 debug 模式，点击 run，发现程序没有停止，说明中断没有被触发。观察板卡，数码管 4 个数字显示正



常，说明前台能正常扫描其他程序。

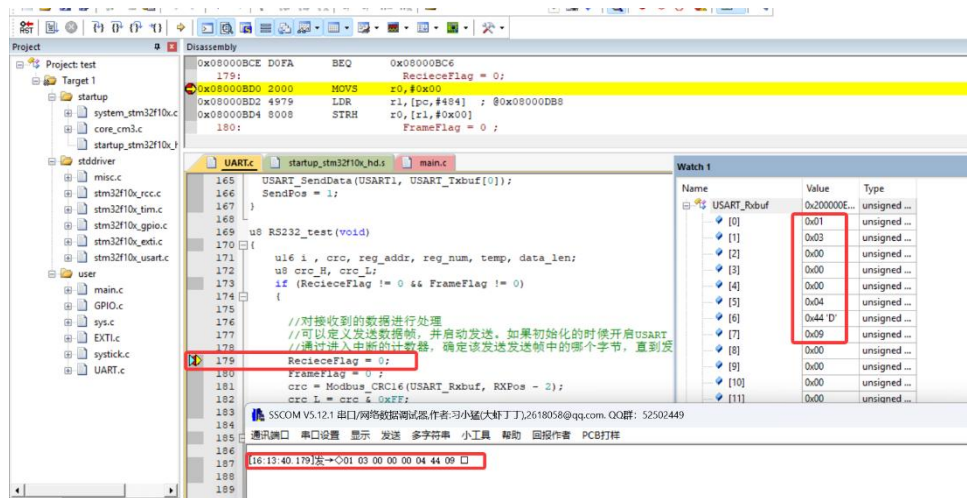
当我们使用 SSCOM 发送“01 03 00 00 00 04（校验码工具自动填充）”，发现程序停止在断点处，观察板卡此时只显示了一个数字，说明进入了中断处理程序，前台无法继续扫描。



综上说明 USART 中断能正常触发。

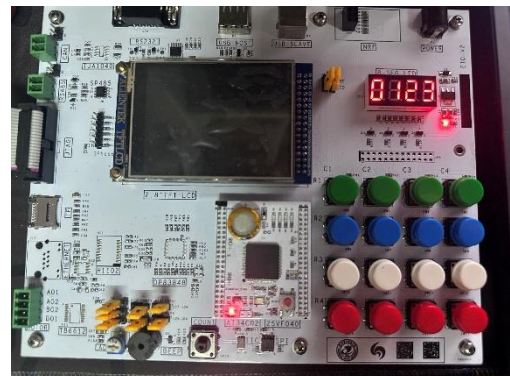
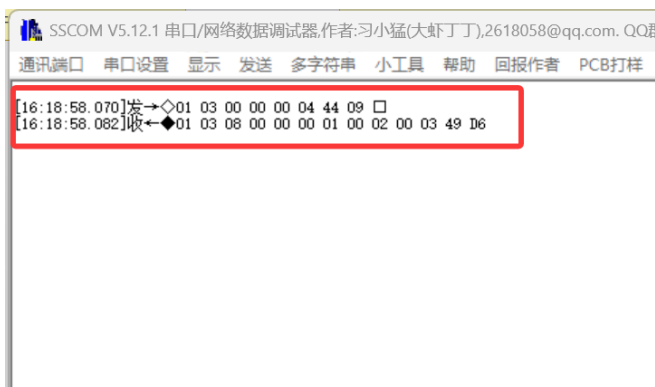
3.2. 检验数据帧的接收

我们在判断接收到数据帧后的第一行程序处打上断点，进入 debug 模式，点击 run，使用 SSCOM 发送一帧数据后程序停在断点处，此时使用 watch 窗口查看 USART_Rxbuf 内容，发现与我们发送的内容一致。说明程序能够正常接受并识别一帧数据。

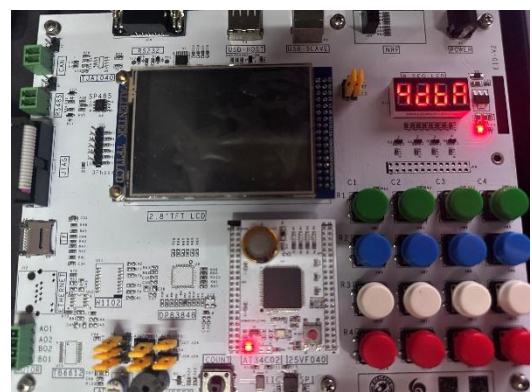
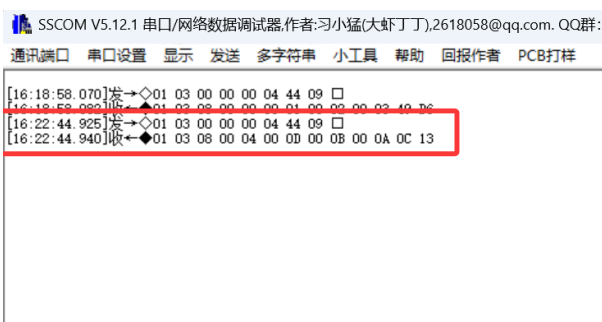


3.3. 检验 Modbus 协议执行

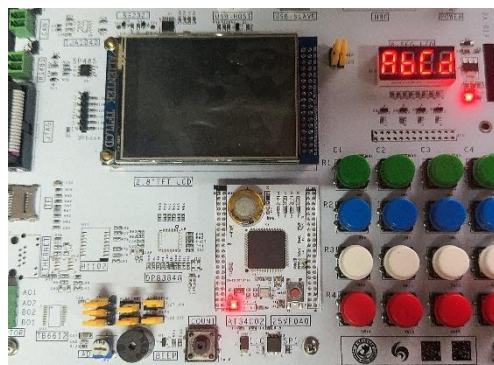
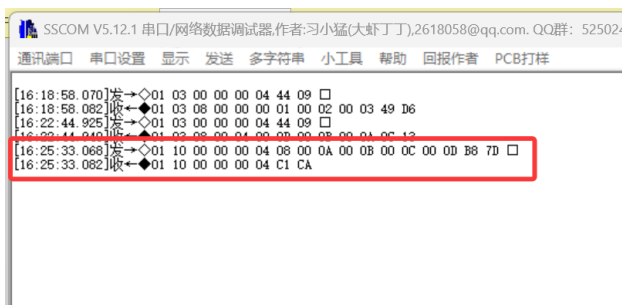
编译并将程序烧入进板卡，按下板卡上的 reset 按钮，让程序开始运行。数码管初始显示“0123”，我们发送读全部 4 个寄存器的指令，SSCOM 收到了板卡的回复消息，告诉我们 0x00~0x03 的值分别为 0、1、2、3，符合预期。



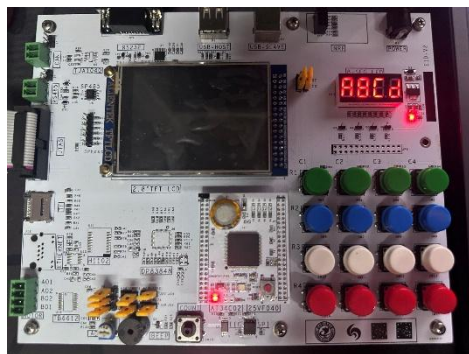
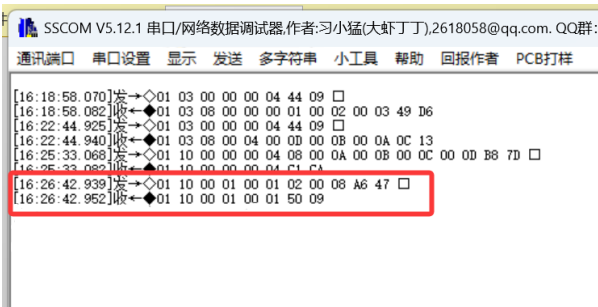
我们再通过按键随机输入一些数，再次发送读全部寄存器的指令，返回的值为 4、D、B、A，与显示内容一致。说明读功能正常。



我们再输入写指令，向四个寄存器分别写入 A、B、C、D。数码管显示内容从“4DBA”变成了“ABCD”，且 SSCOM 也收到板卡回复。



我们再尝试修改第二个数码管从“B”变成“8”。数码管显示内容成功改变。



至此说明 Modbus 的 0x03 与 0x10 功能码实现成功。

4. 心得与体会

本次实验让我把串口通信、协议解析和中断驱动的概念真正串在了一起。在 STM32 上实现 USART 并跑通部分 Modbus 协议后，我对数据帧结构、寄存器映射以及中断方式的发送接收流程有了更直观的理解。前后台配合、中断接力的写法比预想中更高效，也更贴近实际工程做法。整体下来，不只是把功能做出来，更是对嵌入式通信机制多了一层把握。继续做到更完整的协议支持，会让系统的鲁棒性进一步提升。

5. 源代码

见附件压缩包。本实践项目同时开源于：

<https://github.com/Justin-Nickel-Wu/Embedded-Systems-Course-Project>