

lab7 引导装载程序

- lab7 引导装载程序
 - 实验目的
 - 实验器材
 - 硬件
 - 软件
 - 实验原理
 - IAP(in-application programming)
 - Xmodem
 - STM32的RAM和FLASH
 - 连接电路图
 - 实验步骤
 - 连线
 - IAP程序
 - APP程序
 - 运行测试结果：
 - 实验心得

实验目的

- 理解bootloader的一般功能和基本工作原理；
- 掌握调整编译链接参数以形成定制的编译结果的方法。

实验器材

硬件

- STM32F103核心板1块；
- ST-Link 1个；
- CP2102
- 杜邦线
- 按钮1个；
- 面包板1块。

软件

- STM32CubeIDE;
- SecureCRT

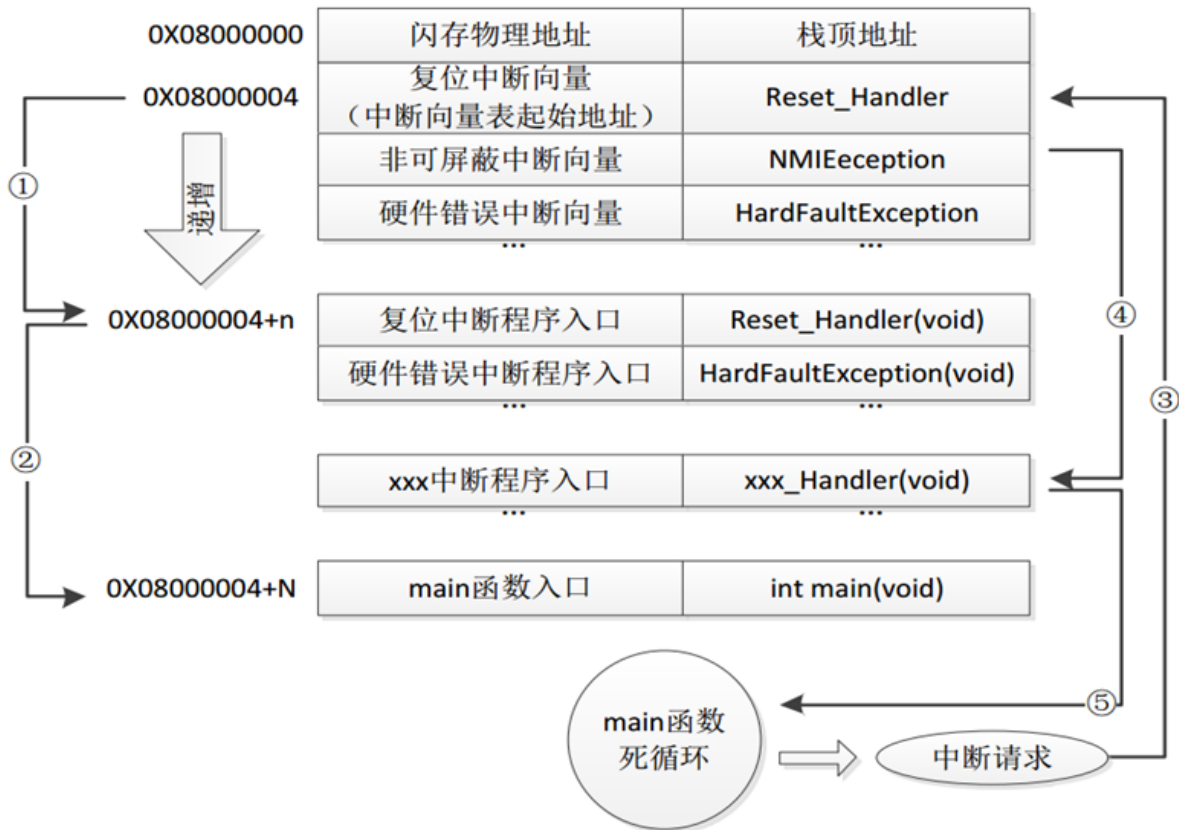
实验原理

IAP(in-application programming)

- IAP，即应用程序内编程。对于大多数基于闪存的系统，一个重要的要求是能够在最终产品中安装固件时进行更新。STM32微控制器可以运行用户特定的固件来对微控制器中嵌入的闪存执行IAP。由于不限制通信接口协议等，只要能通过任意通信接口拿到新版固件包数据（bin文件），就能自己升级固件。这就能

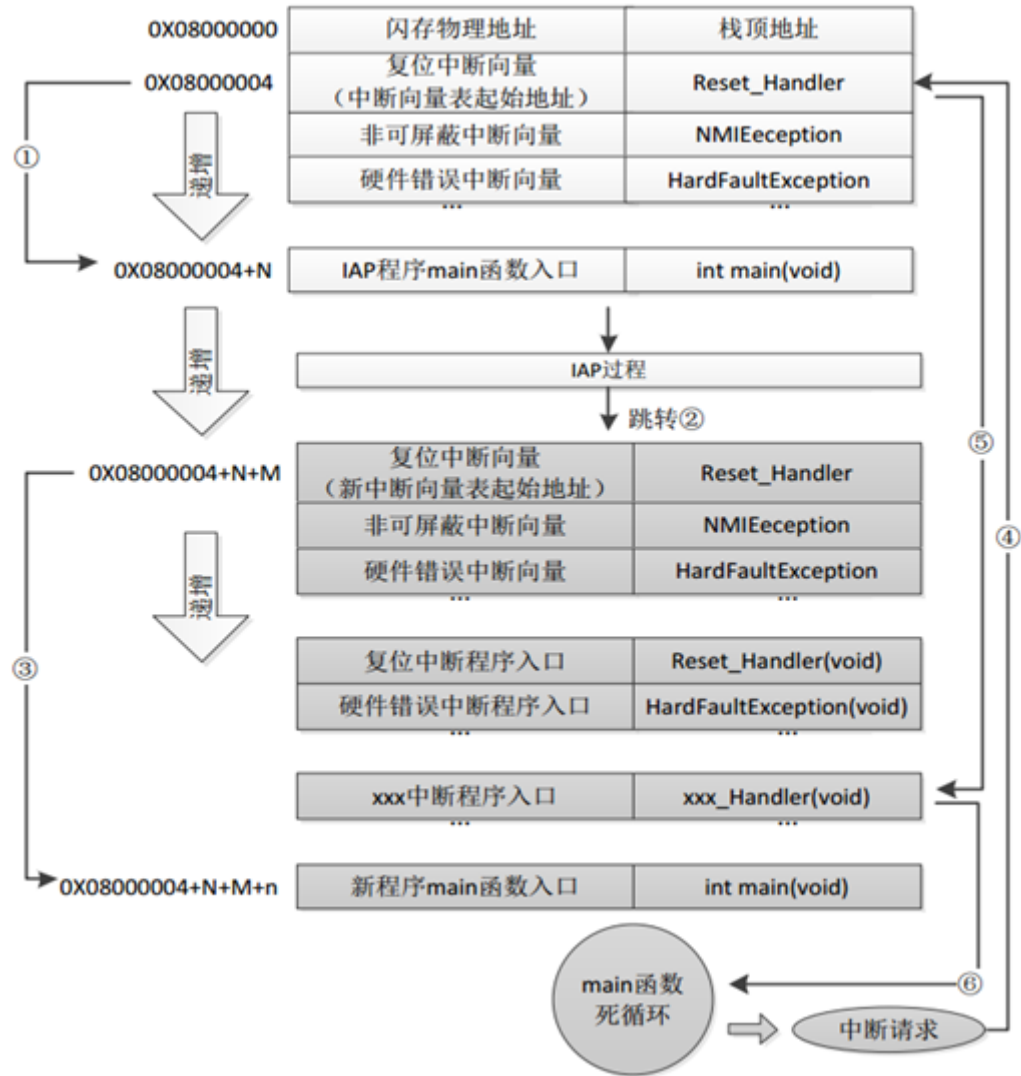
做到添加 外部无线模块（4G模块、wifi）做到OTA升级。也可以使用U盘或TF卡等外部存储设备做到OTG升级。U盘升级的IAP官方有模板程序叫：FWUpgrade_Standalone。

- IAP执行原理
 - 正常程序上电执行流程：



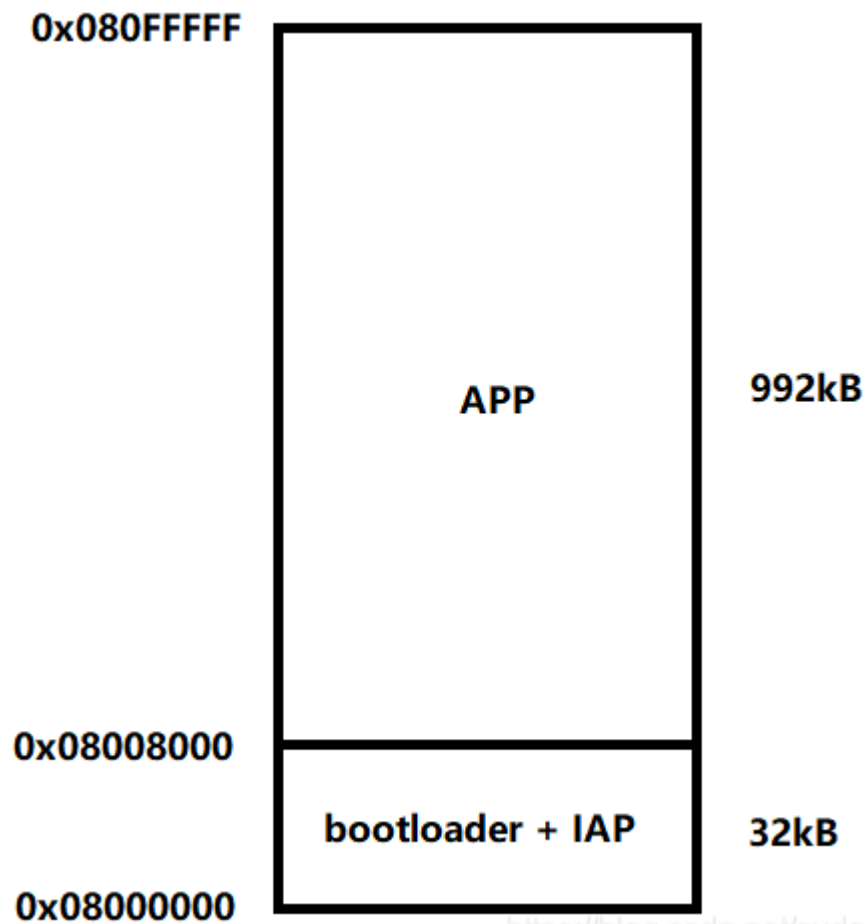
- 从Flash相对0地址即STM32的0x8000000地址开始执行，存储的是栈顶地址。再偏移4个字节找到中断向量表起始地址，即复位中断向量，存储这复位中断程序入口，跳转到复位中断程序入口执行复位中断服务函数Reset_Handle(void)。
- 复位中断服务函数Reset_Handle(void)执行完会跳转到main函数，main函数循环运行。
- main函数死循环时，发生中断请求，然后PC指针会跳转到中断向量表寻找对应的中断向量。
- 找到对应的中断向量后执行对应的中断服务函数xxx_Handler(void)。
- 执行中断服务函数完成后，返回main函数循环运行。

■ 加入IAP后的上电程序执行流程

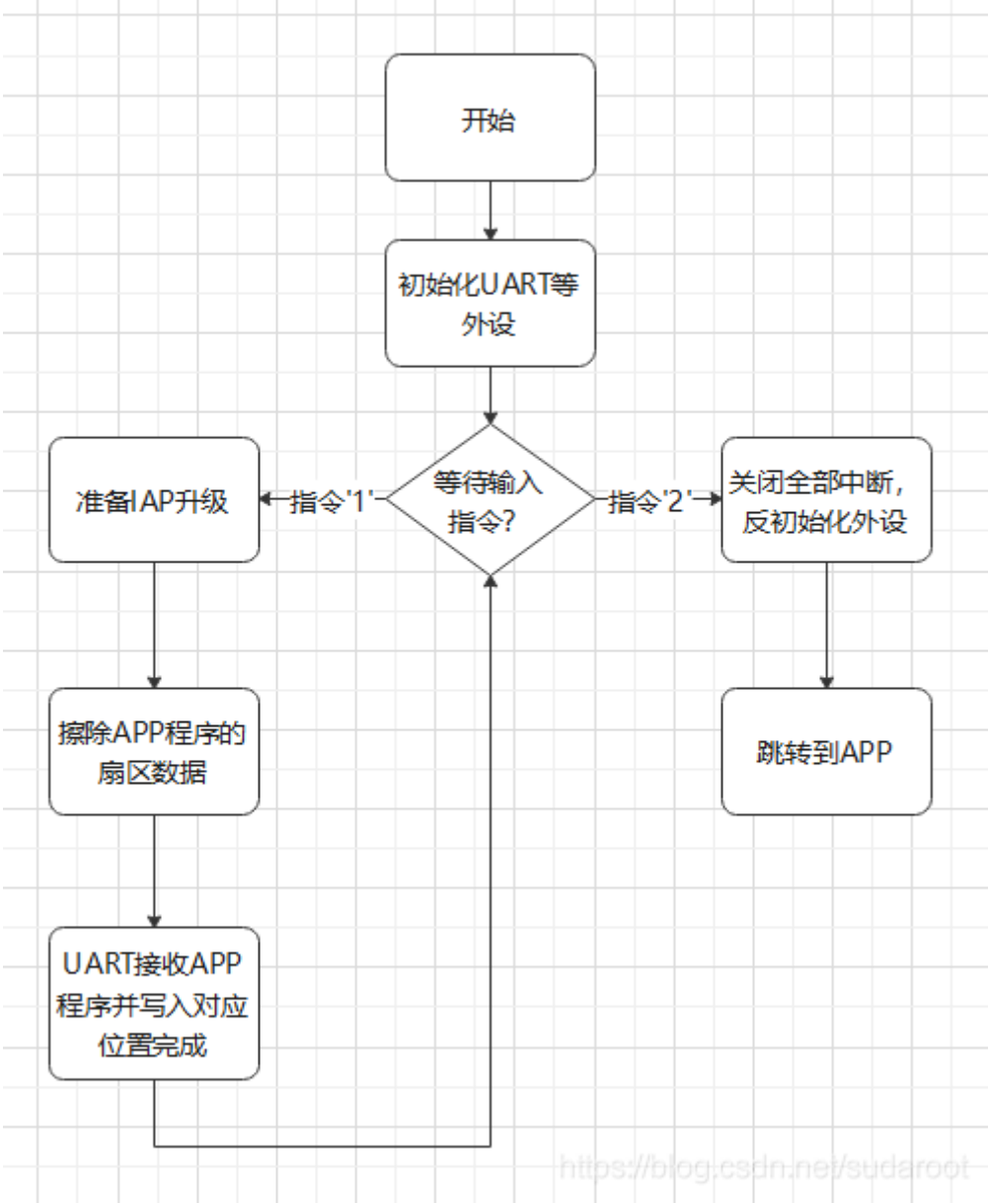


- 加入IAP后其实就把芯片内的Flash分成了两个程序。IAP过程前是一个程序，IAP后是一个程序。
- 由于是两个程序，我们一般会把放在flash相对的0地址的程序叫bootloader引导启动程序，后面的都叫APP用户程序。
- 如上图的IAP程序就占用了flash的M个字节大小，然后通过跳转后APP程序的中断向量表也往后偏移了M个字节。
- bootloader到APP的跳转是通过函数指针跳转。跳转前需要关闭所有的中断后取消外设功能，跳转后需要更新中断向量表。

- 在Flash运行的IAP的简单框图



- 将芯片内部flash分区两个区域，分别存放bootloader和APP程序。



<https://blog.csdn.net/sudaroot>

- IAP工作简单流程图

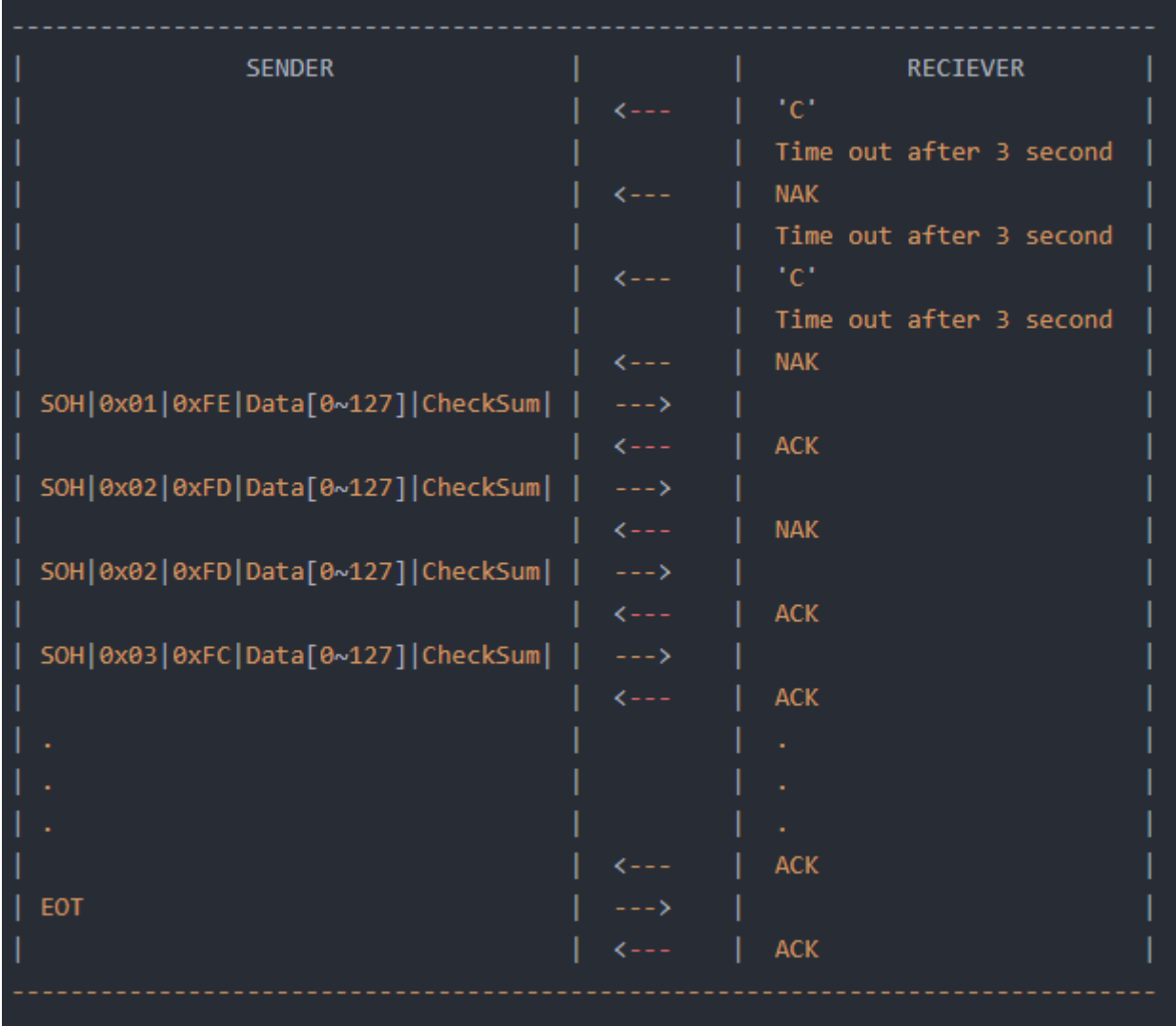
Xmodem

- XModem是一种在串口通信中广泛使用的异步文件传输协议，分为XModem和1k-XModem协议两种，前者使用128字节的数据块，后者使用1024字节即1k字节的数据块。
- XModem信息包格式
 - XModem协议最早由Ward Christensen在20世纪70年代提出并实现的，传输数据单位为信息包，信息包格式如下：

Byte1	Byte2	Byte3	Byte4~Byte131	Byte132-Byte133
Start Of Header	Packet Number	~(Packet Number)	Packet Data	16Bit CRC

- 校验方式
 - CRC16校验

• 传输流程



- 当接收方要求发送方以CRC16校验方式发送数据时以C来请求，发送方对此做出应答，流程就如上图所示。当发送方仅仅支持校验和方式时，则接收方要发送NAK来请求，要求以校验和方式来发送数据，如果仅仅支持CRC16校验方式，则只能发送C来请求。如果两者都支持的话，优先发送C来请求。

STM32的RAM和FLASH

- STM32有两个存储空间，一个是片上的FLASH，一个是片上的RAM。
- 在烧写程序的时候，需要烧写bin文件或者hex文件到STM32的flash中，被烧写的文件可以称为映像文件image。Image的内容包含三部分: code、R0-data和RW-data。
- STM32上电启动后，CPU根据boot0和boot1的硬件引脚决定从FLASH还是RAM中启动，默认是从FLASH中启动，启动之后会搬运RW-data到RAM，但是不会搬运code，也就是说CPU执行的代码是在FLASH中读取的，而不是在RAM中。
- 但是可以通过修改连接配置的Id文件改变code的存放位置。

连接电路图

- CP2102连接

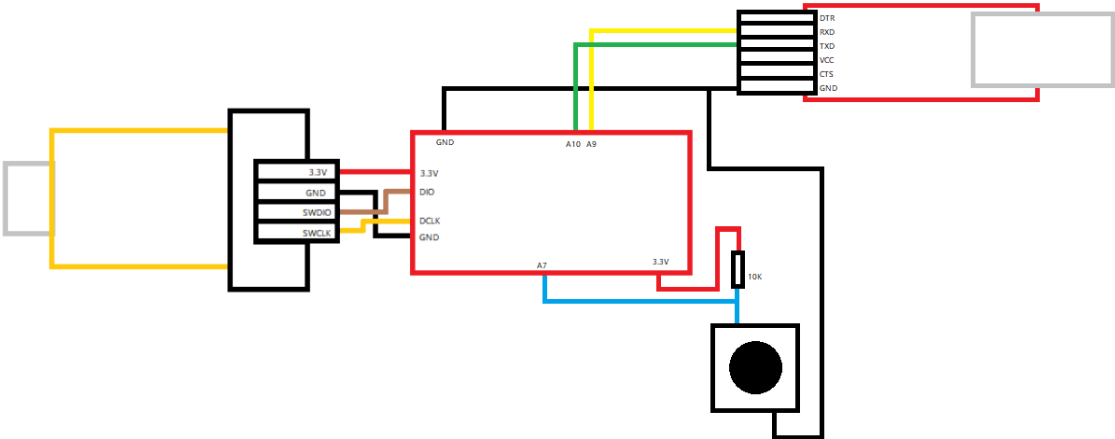
103	CP2102	颜色	意义
A9	RXD	黄色	103发送数据给PC

103	CP2102	颜色	意义
A10	TXD	绿色	PC发送数据给103
GND	GND	黑色	接地

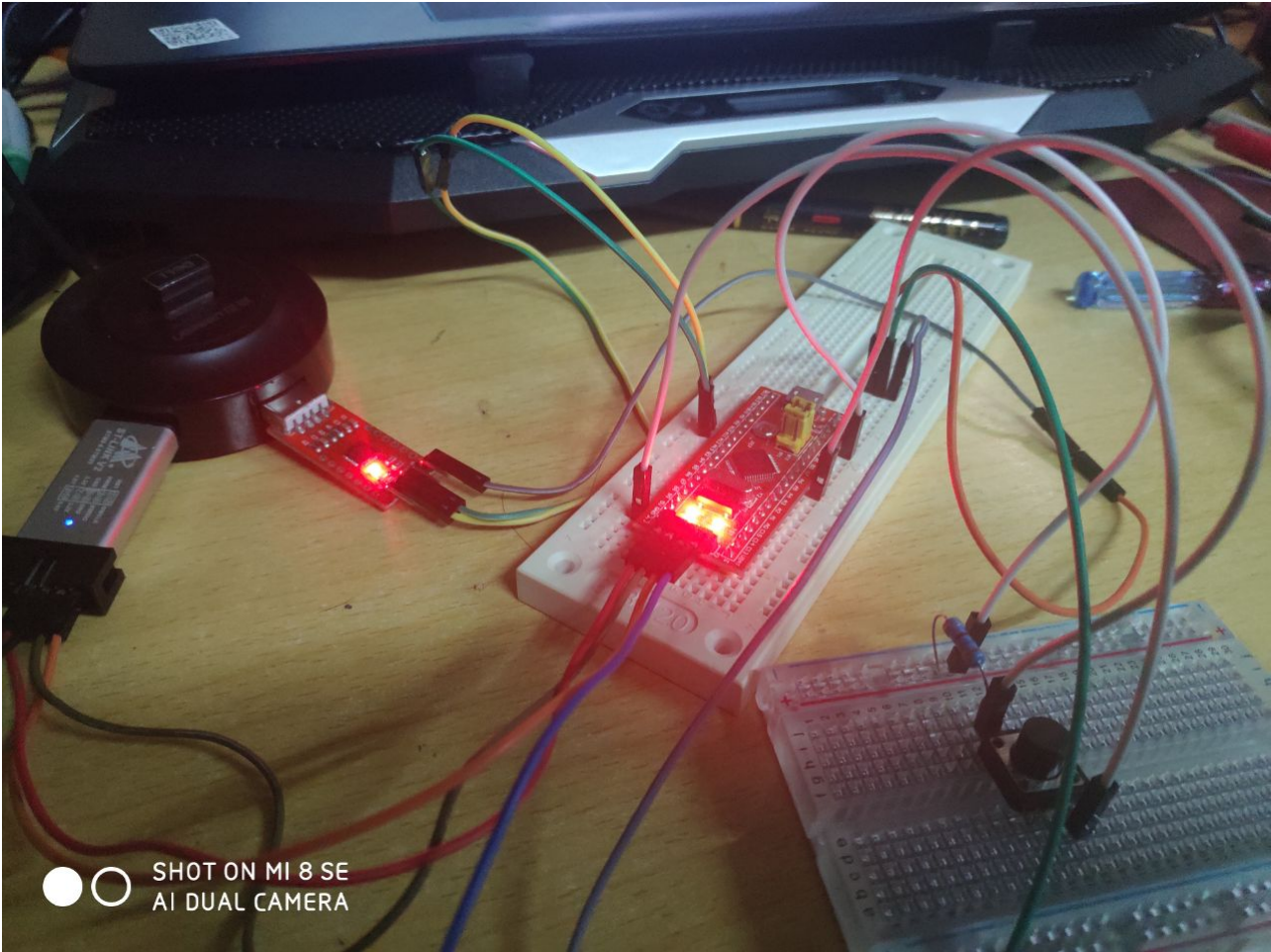
实验步骤

连线

- 连线示意图



- 连线实物图



IAP程序

```

/* ram */
// 写RAM
ram_status ram_write(uint32_t address, uint32_t *data, uint32_t length)
{
    ram_status status = RAM_OK;
    uint32_t i;
    for (i = 0u; (i < length) && (RAM_OK == status); i++)
    {
        // 判断是否越界
        if (RAM_APP_END_ADDRESS <= address)
        {
            status |= RAM_ERROR_SIZE;
        }
        else // 写RAM
        {
            *(__IO uint32_t *) (address) = data[i];
            if (((data[i])) != (*(volatile uint32_t *) address))
            {
                status = HAL_ERROR;
            }
            address += 4u;
        }
    }
    return status;
}

// FLASH中的bootloader跳转到RAM中的APP
void ram_jump_to_app(uint32_t address)
{
    // 函数指针, 指向bootloader将跳转到要运行的APP程序地址
    fnc_ptr jump_to_app;
    jump_to_app = (fnc_ptr) (*(volatile uint32_t *) (address+4u));
    HAL_DeInit();
    __set_MSP(*(volatile uint32_t *) address);
    for(int i = 0; i < 8; i++)
    {
        NVIC->ICER[i] = 0xFFFFFFFF; // 关闭中断
        NVIC->ICPR[i] = 0xFFFFFFFF; // 清除中断标志位
    }
    jump_to_app();
}

/* Xmodem */
/* Xmodem (128 bytes) packet format
 * Byte 0:      Header
 * Byte 1:      Packet number
 * Byte 2:      Packet number complement
 * Bytes 3-130: Data
 * Bytes 131-132: CRC
 */
/* Xmodem (1024 bytes) packet format

```



```

* Byte 0:      Header
* Byte 1:      Packet number
* Byte 2:      Packet number complement
* Bytes 3-1026: Data
* Bytes 1027-1028: CRC
*/
// xmodem接收
void xmodem_receive(uint32_t addr)
{
    volatile xmodem_status status = X_OK;
    uint8_t error_number = 0u;

    x_first_packet_received = false;
    xmodem_packet_number = 1u;
    xmodem_actual_ram_address = addr;

    while (X_OK == status)
    {
        uint8_t header = 0x00u;
        uart_status comm_status = uart_receive(&header, 1u);    // 获得报头
        if ((UART_OK != comm_status) && (false == x_first_packet_received))
        {
            (void)uart_transmit_ch(X_C);    // 使用CRC16-Xmodem
        }
        else if ((UART_OK != comm_status) && (true == x_first_packet_received))
        {
            status = xmodem_error_handler(&error_number, X_MAX_ERRORS);
        }
        /* 报头可以是 SOH, STX, EOT, CAN. */
        switch (header)
        {
            xmodem_status packet_status = X_ERROR;
            case X_SOH:
            case X_STX:
                packet_status = xmodem_handle_packet(header);    // 解包
                if (X_OK == packet_status)    // 解包顺利, 发送ACK
                {
                    (void)uart_transmit_ch(X_ACK);
                }
                else if (X_ERROR_RAM == packet_status)    // 写入RAM异常导致解包异常
                {
                    error_number = X_MAX_ERRORS;
                    status = xmodem_error_handler(&error_number, X_MAX_ERRORS);
                }
                else    // 其他异常
                {
                    status = xmodem_error_handler(&error_number, X_MAX_ERRORS);
                }
                break;
            case X_EOT:    // 传输结束
                (void)uart_transmit_ch(X_ACK);
                (void)uart_transmit_str((uint8_t *)"\n\rEnd of load command. \n\r");
                break;
        }
    }
}

```

```

        case X_CAN:
            status = X_ERROR;
            break;
        default:
            if (UART_OK == comm_status)
            {
                status = xmodem_error_handler(&error_number, X_MAX_ERRORS);
            }
            break;
    }
}

// xmodem 解包
static xmodem_status xmodem_handle_packet(uint8_t header)
{
    xmodem_status status = X_OK;
    uint16_t size = 0u;

    uint8_t received_packet_number[X_PACKET_NUMBER_SIZE];
    uint8_t received_packet_data[X_PACKET_1024_SIZE];
    uint8_t received_packet_crc[X_PACKET_CRC_SIZE];

    if (X_SOH == header)
    {
        size = X_PACKET_128_SIZE;    // 128字节格式
    }
    else if (X_STX == header)
    {
        size = X_PACKET_1024_SIZE;    // 1024字节格式
    }
    else
    {
        status |= X_ERROR;
    }

    uart_status comm_status = UART_OK;
    comm_status |= uart_receive(&received_packet_number[0u],
X_PACKET_NUMBER_SIZE); // 获得包序号
    comm_status |= uart_receive(&received_packet_data[0u], size);    // 获得包数据
    comm_status |= uart_receive(&received_packet_crc[0u], X_PACKET_CRC_SIZE);    //
获得CRC校验
    uint16_t crc_received =
((uint16_t)received_packet_crc[X_PACKET_CRC_HIGH_INDEX] << 8u) |
((uint16_t)received_packet_crc[X_PACKET_CRC_LOW_INDEX]);    // 大小端规则转化
    uint16_t crc_calculated = CalcCRC16(&received_packet_data[0u], size);    // CRC
校验

    if (UART_OK != comm_status)
    {
        status |= X_ERROR_UART;
    }
    if ((X_OK == status) && (false == x_first_packet_received))
    {
        x_first_packet_received = true;
    }
}

```

```

    }
    if (X_OK == status)
    {
        if (xmodem_packet_number != received_packet_number[0u])
        {
            status |= X_ERROR_NUMBER;
        }
        if (255u != (received_packet_number[X_PACKET_NUMBER_INDEX] +
received_packet_number[X_PACKET_NUMBER_COMPLEMENT_INDEX]))
        {
            status |= X_ERROR_NUMBER;
        }
        if (crc_calculated != crc_received)
        {
            status |= X_ERROR_CRC;
        }
    }
    // 写入RAM
    if ((X_OK == status) && (RAM_OK != ram_write(xmodem_actual_ram_address,
(uint32_t *)&received_packet_data[0u], (uint32_t)size / 4u)))
    {
        /* RAM error. */
        status |= X_ERROR_RAM;
    }
    /* Raise the packet number and the address counters (if there weren't any
errors). */
    if (X_OK == status)
    {
        xmodem_packet_number++;
        xmodem_actual_ram_address += size;
    }
    return status;
}
/* main.c */
uint8_t ch;
int number;
uart_transmit_str("Lab is running !\r\n");
while (1)
{
    /* USER CODE END WHILE */
    ch = 0;
    number = 0;
    /* USER CODE BEGIN 3 */
    uint8_t str[100], length;
    uint8_t output[100];
    // 输入指令
    uart_transmit_str("Command\r\n");
    while(1)
    {
        if(uart_receive(&ch, 1) == HAL_OK)
        {
            Receive_Buffer[number++] = ch;
        }
        if(number == BUFFER_SIZE || ch == '\r') // 以回车分割指令

```

```
        {
            recv_end_flag = 1;
            break;
        }
    }
    // 指令分析
    if (recv_end_flag == 1)
    {
        i = 0;
        // 分离指令
        while (Receive_Buffer[i] != ' ')
        {
            Instruction[i] = Receive_Buffer[i];
            i++;
        }
        // peek指令分析
        if (!strcmp(Instruction, "peek"))
        {
            // 获得地址
            j = 7;
            while (j > 0)
            {
                i++;
                if ((Receive_Buffer[i] == '0' && Receive_Buffer[i + 1] == 'x') ||
Receive_Buffer[i] == 'x')
                    continue;
                else
                    Addr[j--] = Receive_Buffer[i];
            }
            Addr[0] = Receive_Buffer[++i];
            // peek
            int addr = Array_to_Hex(Addr);
            sprintf((char *)Data, "%08X\r\n", *((int *)addr));
            uart_transmit_str(Data);
        }
        // poke指令分析
        else if (!strcmp(Instruction, "poke"))
        {
            // 获得地址
            j = 7;
            while (j > 0)
            {
                i++;
                if ((Receive_Buffer[i] == '0' && Receive_Buffer[i + 1] == 'x') ||
Receive_Buffer[i] == 'x')
                    continue;
                else
                    Addr[j--] = Receive_Buffer[i];
            }
            Addr[0] = Receive_Buffer[++i];
            int addr = Array_to_Hex(Addr);
            // 获得数据
            i += 2;
            j = 7;
```

```

        while (j > 0)
        {
            if ((Receive_Buffer[i] == '0' && Receive_Buffer[i + 1] == 'x') ||
Receive_Buffer[i] == 'x')
            {
                i++;
                continue;
            }
            else
                Data[j--] = Receive_Buffer[i];
            i++;
        }
        Data[0] = Receive_Buffer[i++];
        int data = Array_to_Hex(Data);
        // poke
        *((int *)addr) = data;
    }
    // print指令分析
    else if (!strcmp(Instruction, "print"))
    {
        // 获得地址
        j = 7;
        while (j > 0)
        {
            i++;
            if ((Receive_Buffer[i] == '0' && Receive_Buffer[i + 1] == 'x') ||
Receive_Buffer[i] == 'x')
                continue;
            else
                Addr[j--] = Receive_Buffer[i];
        }
        Addr[0] = Receive_Buffer[++i];
        int addr = Array_to_Hex(Addr);
        // print
        sprintf(output, "%s\r\n", addr);
        uart_transmit_str(output);
    }
    // load指令分析
    else if (!strcmp(Instruction, "load"))
    {
        // 获得地址
        j = 7;
        while (j > 0)
        {
            i++;
            if ((Receive_Buffer[i] == '0' && Receive_Buffer[i + 1] == 'x') ||
Receive_Buffer[i] == 'x')
                continue;
            else
                Addr[j--] = Receive_Buffer[i];
        }
        Addr[0] = Receive_Buffer[++i];
        uint32_t addr = Array_to_Hex(Addr);
        uart_transmit_str("Load Begin!\r\n");
    }

```

```

        // load
        xmodem_receive(addr);
    }
    // run指令分析
    else if (!strcmp(Instruction, "run"))
    {
        // 获得地址
        j = 7;
        while (j > 0)
        {
            i++;
            if ((Receive_Buffer[i] == '0' && Receive_Buffer[i + 1] == 'x') ||
                Receive_Buffer[i] == 'x')
                continue;
            else
                Addr[j--] = Receive_Buffer[i];
        }
        Addr[0] = Receive_Buffer[++i];
        uint32_t addr = Array_to_Hex(Addr);
        uart_transmit_str("Run Begin!\r\n");
        // run
        ram_jump_to_app(addr);
    }
    // 还原初始化
    for (i = 0; i < 5; i++)
    {
        Instruction[i] = 0;
    }
    for(i = 0; i < number; i++)
    {
        Receive_Buffer[i] = 0;
    }
    recv_end_flag = 0;
}
}

```

APP程序

```

/* 程序在RAM中运行如果要使用中断的话要涉及到中断向量表的重定位，由于其复杂性，因此重写
lab2中while(1)循环如下： */
count = 0;
if (HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin) == GPIO_PIN_RESET)
{
    while(count != 100)          // 代替HAL_Delay()进行延时
        count++;
    if(HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin) == GPIO_PIN_RESET)\
        BtnState = 1;
    else
        BtnState = 0;
}
else

```

```

    BtnState = 0;
    if(BtnState != last_BtnState && BtnState == 1)
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
    last_BtnState = BtnState;

```

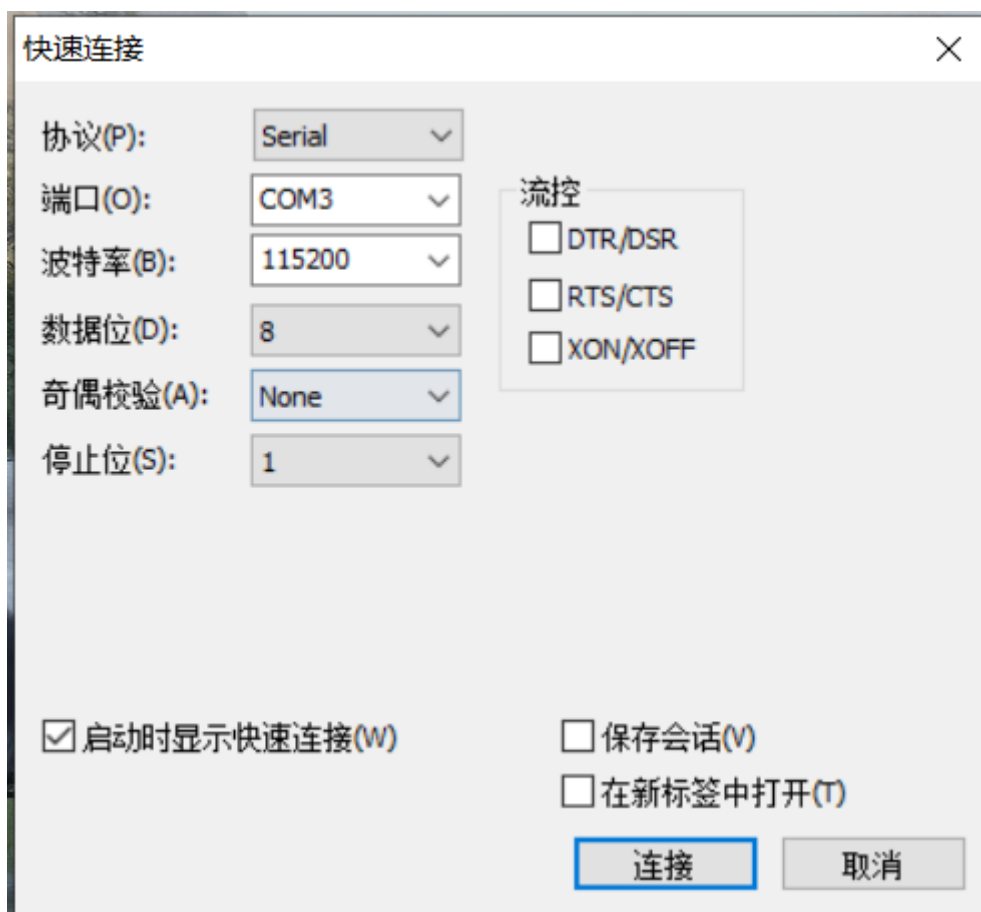
```

/* 原 */
MEMORY
{
    RAM      (xrw)      : ORIGIN = 0x20000000,   LENGTH = 20K
    FLASH    (rx)       : ORIGIN = 0x80000000,   LENGTH = 64K
}
/* 修改后 */
MEMORY
{
    RAM      (xrw)      : ORIGIN = 0x20003000,   LENGTH = 8K
    FLASH    (rx)       : ORIGIN = 0x20001000,   LENGTH = 8K
}

```

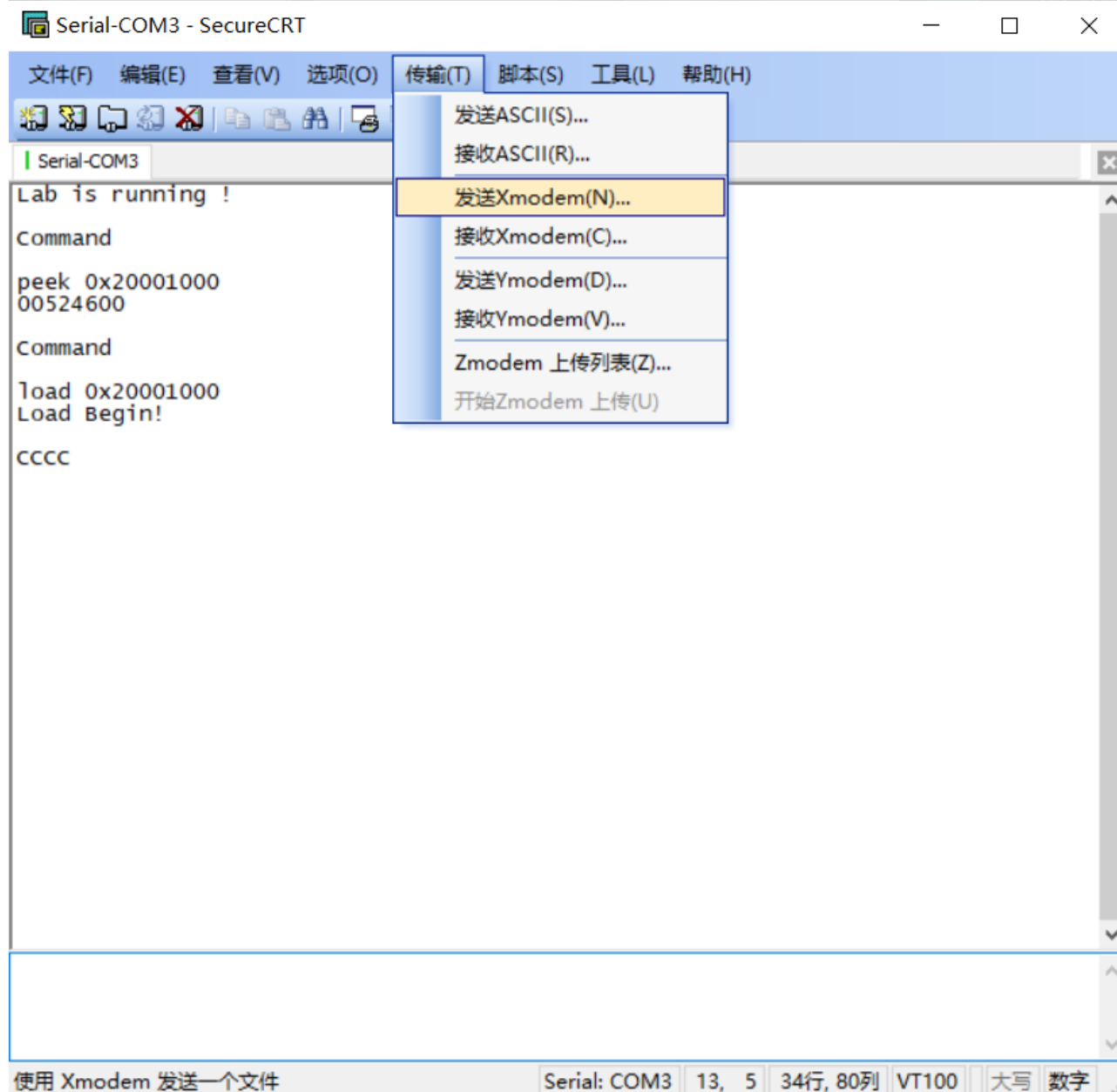
/* 要让程序在RAM上运行，则只需要对其链接文件进行修改。最简单的方法就是对地址做修改。可以看出RAM仅有20K大小的空间去运行APP程序，经过实验发现有大约4K的空间bootloader程序需要占用，剩余16K用来存储运行APP程序。由于不涉及到中断向量表重定位等复杂操作，因此APP程序越简单越好 */

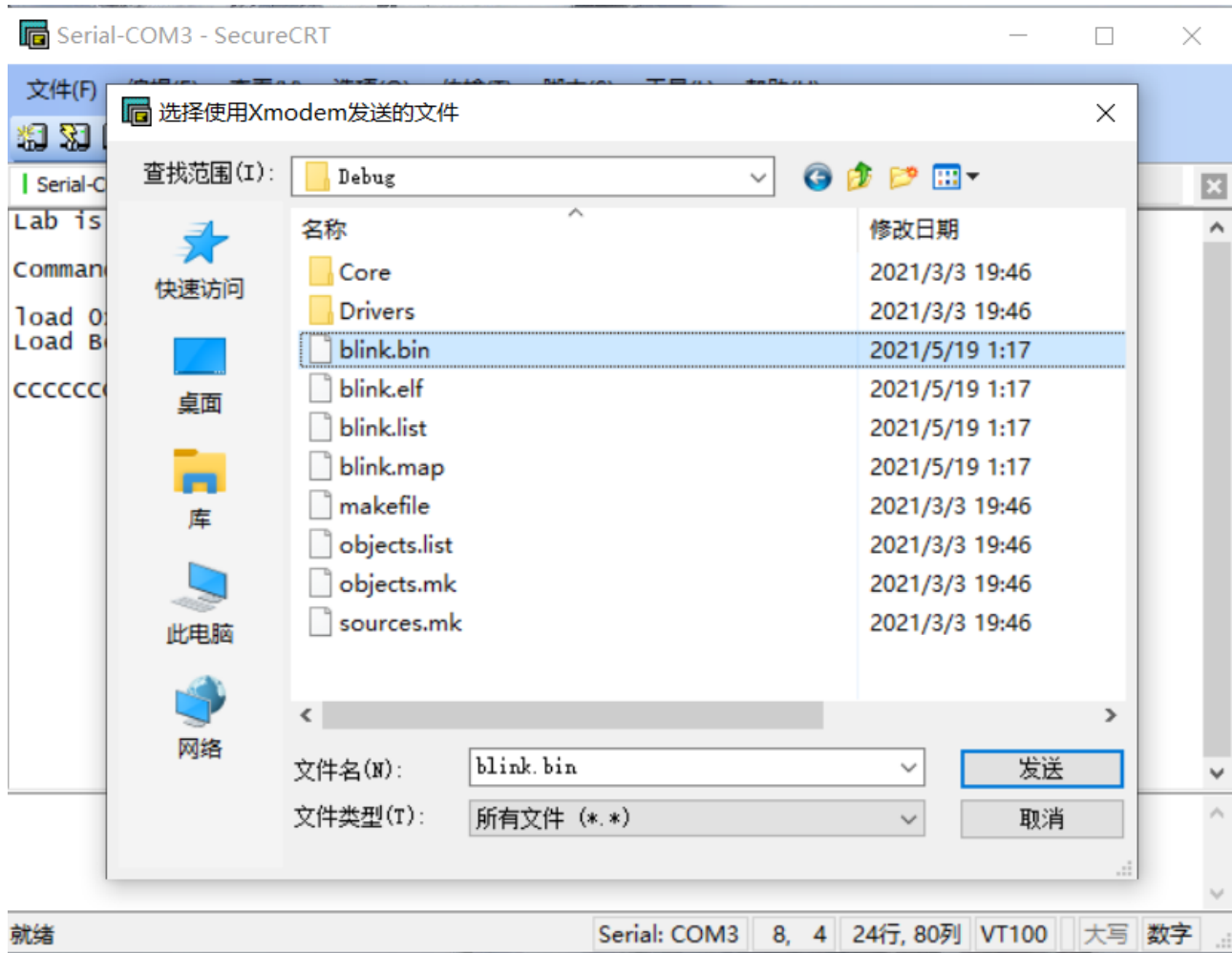
运行测试结果：



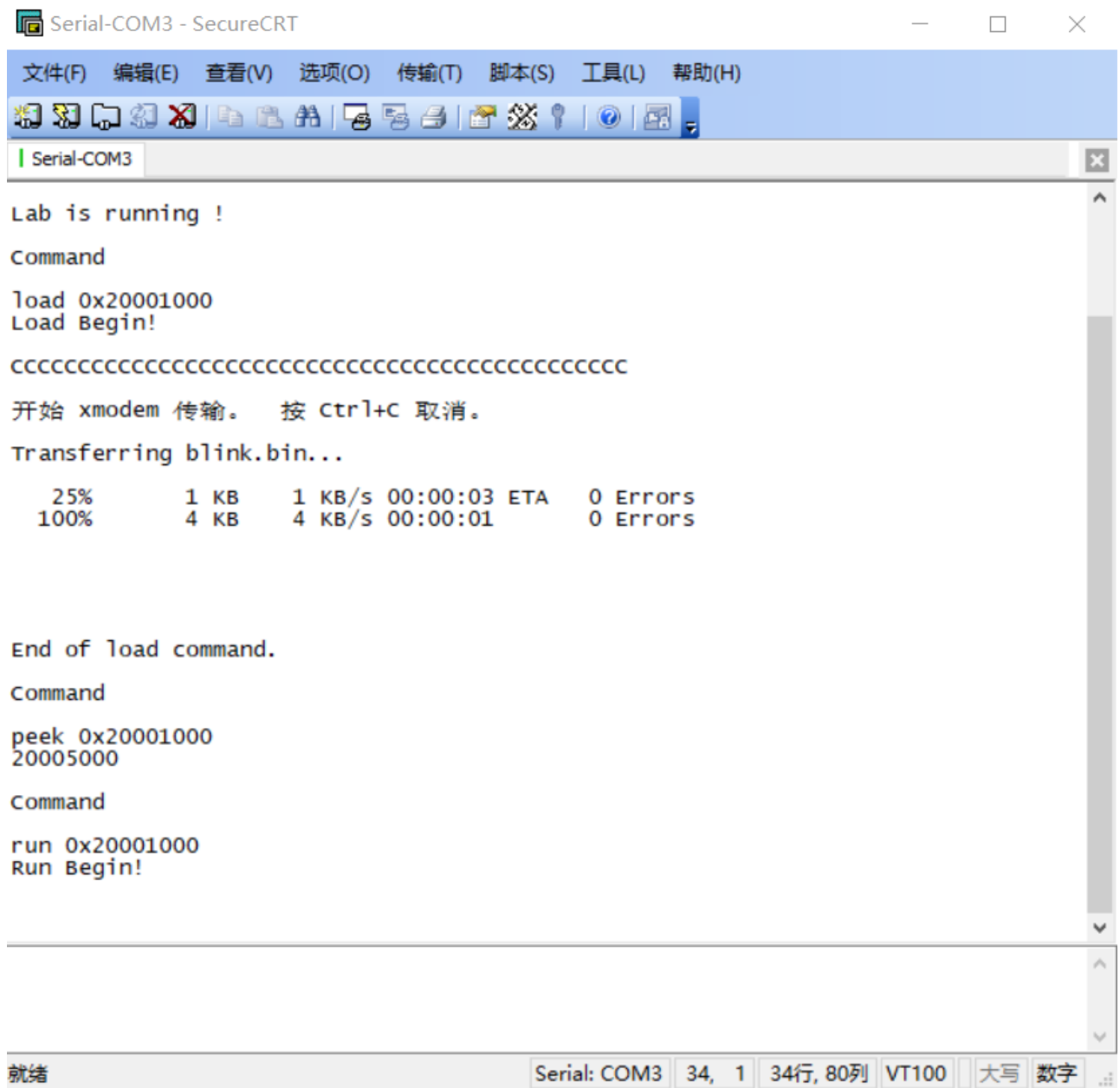
- SecureCRT配置

- 键入load指令后，使用SecureCRT以Xmodem协议发送.bin文件





- 键入`peek`指令查看.bin文件是否写入。键入`run`运行APP程序

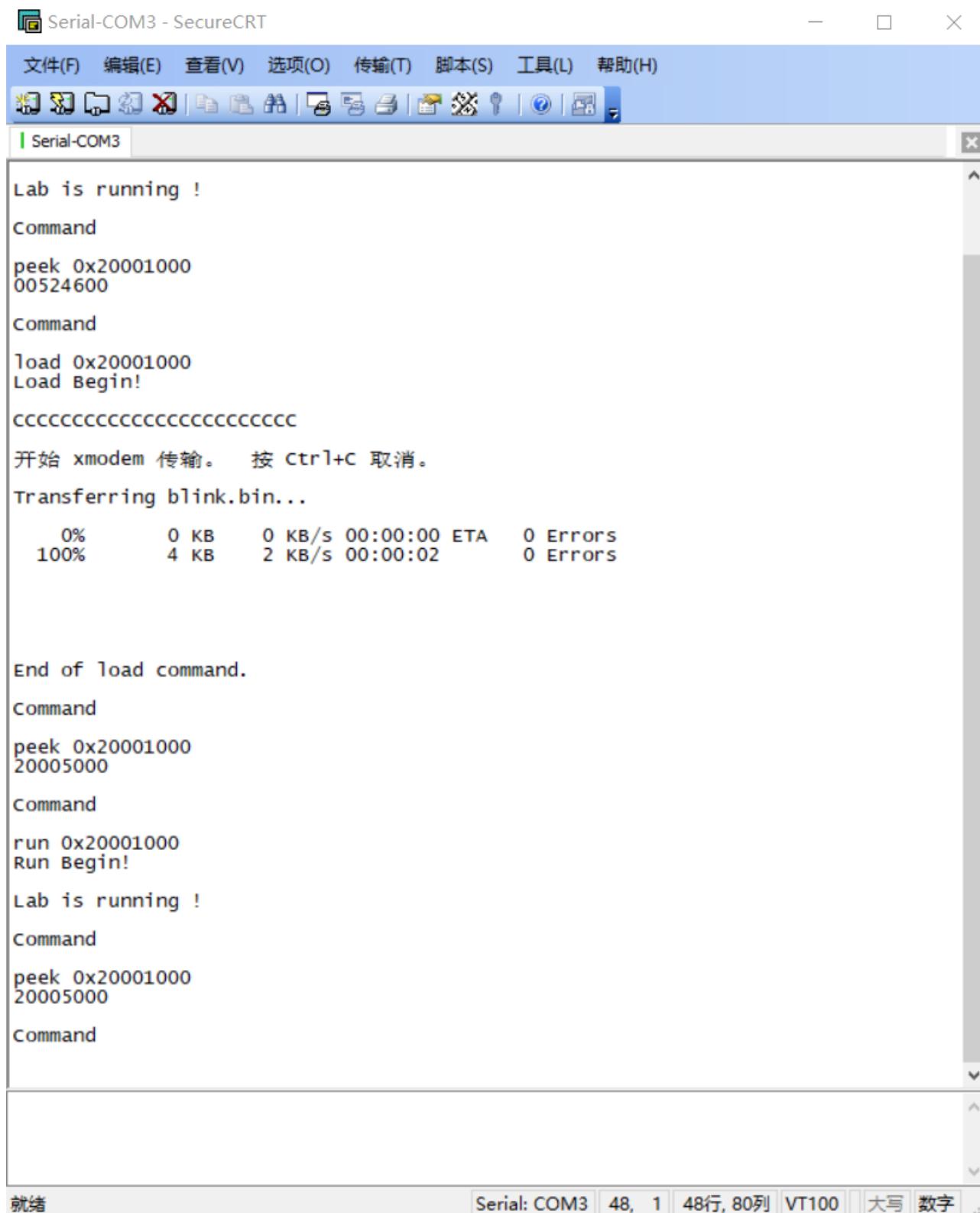


```
Serial-COM3 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM3
Lab is running !
Command
load 0x20001000
Load Begin!
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
开始 xmodem 传输。 按 ctrl+c 取消。
Transferring blink.bin...
    25%      1 KB      1 KB/s 00:00:03 ETA    0 Errors
   100%      4 KB      4 KB/s 00:00:01      0 Errors

End of load command.
Command
peek 0x20001000
20005000
Command
run 0x20001000
Run Begin!

就绪 Serial: COM3 34, 1 34行, 80列 VT100 大写 数字
```

- 结果是运行成功
- 按下板子上的`reset`按钮重启后，键入`peek`指令读取写入APP程序的位置，发现程序依然存在，所以可以确定APP程序写入了RAM中。



```
Serial-COM3 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM3
Lab is running !
Command
peek 0x20001000
00524600
Command
load 0x20001000
Load Begin!
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
开始 xmodem 传输。 按 Ctrl+C 取消。
Transferring blink.bin...
    0%      0 KB      0 KB/s 00:00:00 ETA    0 Errors
  100%     4 KB      2 KB/s 00:00:02      0 Errors

End of load command.
Command
peek 0x20001000
20005000
Command
run 0x20001000
Run Begin!
Lab is running !
Command
peek 0x20001000
20005000
Command

就绪 Serial: COM3 48, 1 48行, 80列 VT100 大写 数字
```

实验心得

- 本次实验可以说是难度最大的一个实验，需要了解的内容非常繁杂，而方向也比较模糊。实验后总结主要有以下几点
 - IAP的概念和原理
 - Xmodem协议或者Zmodem协议
 - RAM和FLASH与链接的关系
- 经过这次实验，对STM32的工作原理、IAP升级以及两种协议有了较深的认识。比较遗憾的是自己尝试写的Zmodem协议没能成功运行，由于调试无从下手，加上时间紧迫，无奈放弃修改。查阅资料使用了

Xmodem协议。不过对协议工作原理的学习应该会有助于此后大作业中对Kiss-Modem等协议的使用。