第3章 串口开发

CHAPTER 3

作为 MCU 的重要外部接口，串口同时也是软件开发重要的调试手段。通用同步异步收发器(USART)能够灵活地与外部设备进行全双工数据交换，满足外部设备对工业标准 NRZ 异步串行数据格式的要求。USART 通过小数波特率发生器提供了多种波特率，通过配置多个缓冲区使用 DMA 可实现高速数据通信。

串口主要有轮询、中断、DMA 三种通信方式，本章逐一介绍其相关原理并通过实例帮助读者掌握串口开发能力。


视频 7

3.1 串口通信：轮询

学习目标

了解串口的工作原理、ARM Cortex-M 系列芯片的串口的分类，通过配置 STM32F407 芯片的串口外设来实现数据的收发。

3.1.1 开发原理

任何 USART 双向通信均需要至少两个引脚：接收数据输入引脚(RX)和发送数据输出引脚(TX)。在同步模式下，需要使用时钟输出引脚(CK)，在硬件流控制模式下，需要使用清除发送引脚(CTS)和发送请求引脚(RTS)。

1. STM32 中串口通信数据包

串口通信的数据包由发送设备通过自身的 TXD 接口传输到接收设备的 RXD 接口，通信双方的数据包格式要一致才能正常收发数据，STM32 中串口通信数据包的内容有起始位、数据位、奇偶校验位、停止位。

1) 起始位和停止位

串口通信的一个数据包从起始信号开始，直到停止信号结束。数据包的起始信号由一个逻辑 0 的数据位表示，而数据包的停止信号可由 0.5、1、1.5 或 2 个逻辑 1 的数据位表示，只要双方约定一致即可。

2) 数据位

在数据包的起始位之后紧接着的就是要传输的主体数据内容，也称为有效数据，有效数据的长度常被约定为 8 或 9 位长。

3) 奇偶校验位

在有效数据之后，有一个可选的数据校验位。由于数据通信相对更容易受到外部干扰导致传输数据出现偏差，可以在传输过程加上校验位来解决这个问题。有无奇偶校验位数据帧

格式如表 3-1 所示。

表 3-1　奇偶校验位

数 据 长 度	奇 偶 校 验	数据帧格式
8	无	起始位＋8 位数据＋停止位
8	有	起始位＋7 位数据＋校验位＋停止位
9	无	起始位＋9 位数据＋停止位
9	有	起始位＋8 位数据＋校验位＋停止位

奇校验是指有效数据和校验位中 1 的个数为奇数，比如一个 8 位字长的有效数据为 00110101，此数据总共有 4 个 1，为达到奇校验效果，校验位为 1，最后传输的数据将是 8 位的有效数据加上 1 位的校验位总共 9 位。

偶校验与奇校验要求刚好相反，要求帧数据和校验位中 1 的个数为偶数，比如 00110101，此数据帧 1 的个数为 4 个，所以偶校验位为 0。

4）波特率

本节主要讲解的是串口异步通信，异步通信中由于没有时钟信号，所以两个通信设备之间约定好波特率，即每个码元的长度，以便对信号进行解码，常见的波特率为 4800bps、9600bps、57600bps、115200bps 等。

USART 功能框图如图 3-1 所示。

5）功能引脚

- TX：发送数据输出引脚。
- RX：接收数据输入引脚。
- SW_RX：数据接收引脚，只用于单线和智能卡模式，属于内部引脚没有具体外部引脚。
- nRTS：发送数据请求引脚，用于指示 UART 已经准备好接收数据。低电平有效，该引脚只适用于硬件流控制。
- nCTS：清除发送引脚，用于在当前传输结束时阻止数据发送。低电平有效，该引脚只适用于硬件流控制。

STM32F407ZGT6 芯片的 USART 引脚分布如图 3-2 所示。

STM32F407ZGT6 有 4 个 USART 和 2 个 UART，其中 USART1 和 USART6 的时钟来源于 APB2 总线时钟，其最大频率为 84MHz，其他 4 个的时钟来源于 APB1 总线时钟，其最大频率为 42MHz。

UART 只是异步传输功能，所以没有 SCLK、nCTS 和 nRTS 功能引脚。

6）数据寄存器

USART_DR 包含了已发送的数据或者接收到的数据。USART_DR 实际上包含了两个寄存器：一个专门用于发送的可写 TDR，另一个专门用于接收的可读 RDR。当进行发送操作时，向 USART_DR 写入的数据会自动存储在 TDR 内；当进行读取操作时，从 USART_DR 读取数据时会自动提取 RDR 数据。

TDR 和 RDR 都是介于系统总线和移位寄存器之间。串行通信是一个位一个位传输的，发送时把 TDR 的内容转移到发送移位寄存器，然后把移位寄存器数据每一位发送出去，接收时把接收到的每一位数据保存在接收移位寄存器内，然后才转移到 RDR。

7）控制器

USART 有专门控制发送的发送器、控制接收的接收器，还有唤醒单元、中断控制等。使用 USART 之前需要将 USART_CR1 寄存器的 UE 位置 1 使能 USART。发送或者接收数据

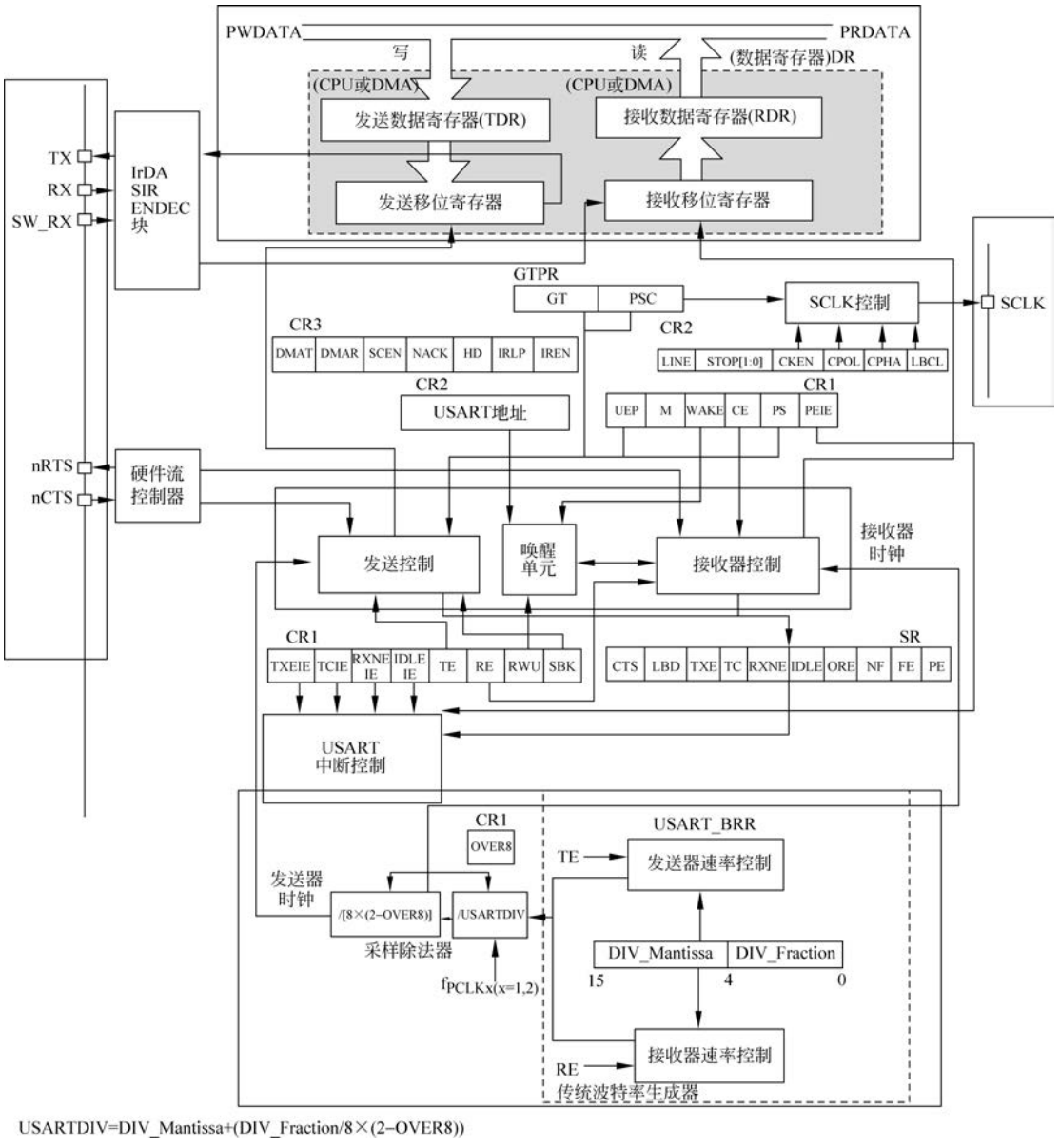

图 3-1　USART 功能框图

	APB2（最高 84MHz）		APB1（最高 42MHz）			
	USART1	**USART6**	**USART2**	**USART3**	**USART4**	**USART5**
TX	PA9/PB6	PC6/PG14	PA2/PD5	PB10/PD8/PC10	PA0/PC10	PC12
RX	PA10/PB7	PC7/PG9	PA3/PD6	PB11/PD9/PC11	PA1/PC11	PD2
SCLK	PA8	PG7/PC8	PA4/PD7	PB12/PD10/PC12	—	—
nCTS	PA11	PG13/PG15	PA0/PD3	PB13/PD11	—	—
nRTS	PA12	PG8/PG12	PA1/PD4	PB14/PD12	—	—

图 3-2　USART 引脚分布

字长可选 8 位或 9 位，由 USART_CR1 的 M 位控制。

8）波特率生成

波特率指数据信号对载波的调制速率，它用单位时间内载波调制状态改变次数来表示。比特率指单位时间内传输的比特数。对于 USART 波特率与比特率相等，波特率越大，传输速率越快。波特率的常用值有 2400bps、9600bps、19200bps、115200bps。

串口通信有 3 种方式用来处理数据，包括轮询方式、中断方式、DMA 方式。本节只介绍轮询方式。轮询方式是指程序中循环查询串口寄存器，如果寄存器接收到数据，则进行相应的数据处理。

2. 寄存器介绍

（1）状态寄存器（USART_SR），如图 3-3 所示。

Address offset: 0x00

Reset value: 0x00C0 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

图 3-3　状态寄存器

- 位 7：传输寄存器为空。当 TDR 寄存器内容被传输到移位寄存器时，硬件将设置此位。
- 位 6：传输完成。当包含数据帧传输完成，且设置了 TXE，硬件将设置此位。
- 位 5：读取数据寄存器非空。当 RDR 移位寄存器的内容转移到 USART_DR 寄存器时，硬件将设置此位。

（2）数据寄存器（USART_DR）。

位 8:0：包含接收或传输的数据字符。

（3）波特率寄存器（USART_BRR）。

位 15:0：用来配置波特率。

（4）控制寄存器 1（USART_CR1），如图 3-4 所示。

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8		UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 3-4　控制寄存器 1

- 位 13：串口使能位。
- 位 12：用来配置数据长度。
- 位 10：使能奇偶校验。
- 位 9：配置奇偶校验。
- 位 3：发送使能。
- 位 2：接收使能。

(5) 控制寄存器 2(USART_CR2),如图 3-5 所示。

Address offset: 0x10

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	Res.	ADD[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

图 3-5　控制寄存器 2

- 位 13:12: 用来配置停止位。
- 位 3:0: 用来配置串口地址。

3. 串口通信配置步骤

(1) 初始化串口和 GPIO 时钟。

(2) 初始化串口引脚 GPIO。

(3) 初始化串口地址、数据长度、停止位、波特率等参数。

(4) 使能串口。

(5) 调用数据收发函数。

3.1.2　开发步骤

(1) 定义结构体变量 UART1_Handler,用来传入串口参数,以便初始化串口。

(2) 定义 UART_Init()函数。

在 UART_Init()函数中配置结构体变量 UART1_Handler 中的串口地址、波特率等参数,最后将结构体变量 UART1_Handler 传入 HAL_UART_Init,以便初始化串口。

```
//初始化串口函数
void UART_Init(void)
{
    UART1_Handler.Instance = USART1;                    //串口 1
    UART1_Handler.Init.BaudRate = 115200;               //波特率
    UART1_Handler.Init.HwFlowCtl = UART_HWCONTROL_NONE; //无硬件流控
    UART1_Handler.Init.Mode = UART_MODE_TX_RX;          //收发模式
    UART1_Handler.Init.Parity = UART_PARITY_NONE;       //无奇偶校验
    UART1_Handler.Init.StopBits = UART_STOPBITS_1;      //一个停止位
    UART1_Handler.Init.WordLength = UART_WORDLENGTH_8B; //字长为 8 位格式

    HAL_UART_Init(&UART1_Handler); //初始化串口
}
```

(3) 重定义 HAL_UART_MspInit()函数初始化,初始化串口硬件参数。

当调用 HAL_UART_Init()串口初始化函数时,该函数内部会调用 HAL_UART_MspInit()函数,用来初始化串口使用到的引脚。这实际是 HAL 库的一个优点,它通过开放一个回调函数 HAL_UART_MspInit(),让用户自己去编写与串口有关的硬件初始化,而与串口相关的参数配置则放在 HAL_UART_Init()函数中,这样在将程序移植到其他 STM32 平台时,只需要修改 HAL_UART_MspInit()函数中的配置即可。

```
//初始化硬件
void HAL_UART_MspInit(UART_HandleTypeDef * huart)
{
```

```
    if(huart->Instance == USART1)
    {
        GPIO_InitTypeDef GPIO_Initure;

        __HAL_RCC_GPIOA_CLK_ENABLE();                    //使能 GPIO 引脚
        __HAL_RCC_USART1_CLK_ENABLE();                   //使能串口

        GPIO_Initure.Mode = GPIO_MODE_AF_PP;             //复用推挽模式
        GPIO_Initure.Pin = GPIO_PIN_9|GPIO_PIN_10;       //PA9、PA10
        GPIO_Initure.Pull = GPIO_PULLUP;                 //上拉
        GPIO_Initure.Speed = GPIO_SPEED_FAST;            //高速
        GPIO_Initure.Alternate = GPIO_AF7_USART1;        //复用为 USART1
        HAL_GPIO_Init(GPIOA, &GPIO_Initure);             //初始化 PA9、PA10
    }
}
```

(4) 定义发送数据函数。函数传入参数为发送数据缓冲区地址和发送数据量的大小。

```
//发送数据函数
void UART_Transmit(uint8_t *pData, uint16_t Size)
{
    HAL_UART_Transmit(&UART1_Handler, pData, Size, HAL_MAX_DELAY);
}
```

(5) 定义接收数据函数。函数传入参数为接收数据缓冲区地址和接收数据量的大小。

```
//接收数据函数
void UART_Receive(uint8_t *pData, uint16_t Size)
{
    HAL_UART_Receive(&UART1_Handler, pData, Size, HAL_MAX_DELAY);
}
```

(6) 添加 printf()的重定向函数,在工程中调用 printf()函数,将通过串口发送数据。

```
//标准库需要的支持函数
struct __FILE
{
    int handle;
};
FILE __stdout;
//重定义 fputc 函数
int fputc(int ch, FILE *f)
{
    while((USART1->SR&0X40)==0);                         //循环发送,直到发送完毕
    USART1->DR = (uint8_t)ch;
    return ch;
}
```

(7) main.c 主函数代码如下:

第一步,定义存储数据变量。

第二步,初始化系统时钟和串口。

第三步,调用数据收发函数进行数据通信。

```
uint8_t buffer;

int main(void)
{
        CLOCLK_Init();                                   //配置系统时钟为 168MHz
        UART_Init();                                     //初始化
```

```
        while(1)
        {
            UART_Receive(&buffer, 1);
            UART_Transmit(&buffer, 1);
        }
}
```

(8) USB 转 TTL 模块连接方法：

- 模块 TX 连接开发板 U1_Rx 引脚；
- 模块 RX 连接开发板 U1_Tx 引脚；
- 模块 GND 连接开发板 GND。

3.1.3　运行结果

将程序下载到开发板中，找到 USART1 引脚通过 USB 转串口模块连接到计算机上，打开串口助手，配置好波特率、数据位、校验位、停止位等，单击打开串口。如果此刻发送一个 1，则会看到串口助手会打印 1，如图 3-6 所示。

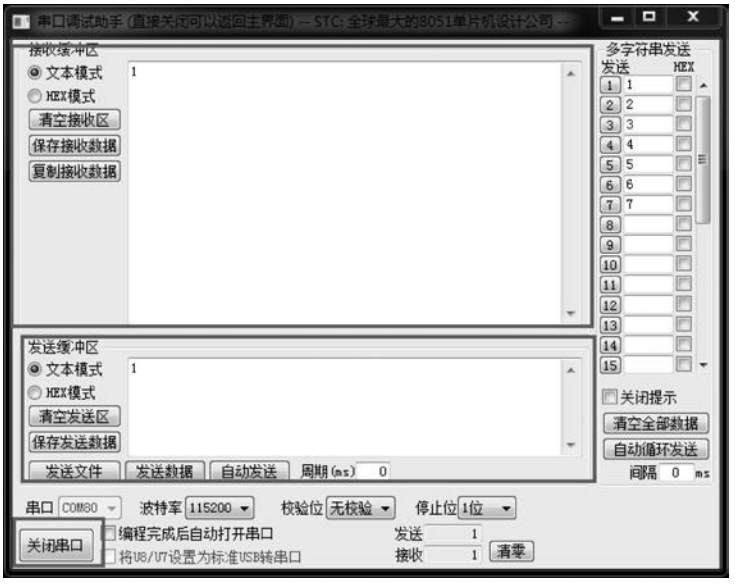

图 3-6　输出结果

练习

(1) 简述 STM32 中串口通信数据包的内容。

(2) 简述串口通信配置步骤。

(3) 将波特率更改为 9600bps，进行串口数据收发。

3.2　串口通信：中断


视频 8

学习目标

了解串口的工作原理、ARM Cortex-M 系列芯片串口的分类以及串口中断响应，通过配置 STM32F407 芯片的串口外设和中断，来实现数据的收发。

3.2.1 开发原理

3.1 节介绍了串口的轮询方式通信，轮询方式主要特点是让 CPU 以一定的周期查询串口外设，如果有数据输入则进行数据处理，否则进行循环判断。轮询方式缺点就是当程序执行内容较多时实时性不高，当数据输入比较频繁的时候，串口不能及时地进行数据处理。针对这个问题，我们引入串口中断通信。

串口中断请求如表 3-2 所示。

表 3-2 串口中断请求

中断事件	事件标志	使能控制位
发送寄存器为空	TXE	TXEIE
清除发送标志	CTS	CTSIE
发送完成	TC	TCIE
接收寄存器为空	RXNE	RXNEIE
溢出错误检测	ORE	
空闲总线检测	IDLE	IDLEIE
校验错误	PE	PEIE
中断标志	LBD	LBDIE
多缓冲区通信中的噪声标记、溢出错误和帧错误	NF 或 ORE 或 FE	EIE

串口中断事件连接在相同的中断向量，如果设置了相应的使能位，那么这些事件将生成一个中断，如图 3-7 所示。

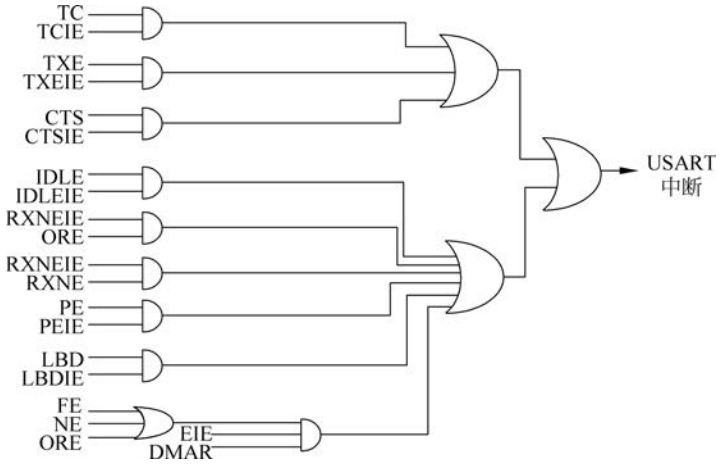

图 3-7 串口中断事件

3.2.2 开发步骤

(1) 定义结构体变量 UART1_Handler 和接收数据变量 buffer。

```
UART_HandleTypeDef UART1_Handler;          //UART 句柄
uint8_t buffer[50] = {0};                  //串口接收数据
```

(2) 定义 UART_Init()函数，在 UART_Init()函数中配置结构体变量 UART1_Handler 中的串口地址、波特率等参数，然后将结构体变量 UART1_Handler 传入到 HAL_UART_Init，以便初始化串口。

```
//初始化串口函数
void UART_Init(void)
{
    UART1_Handler.Instance = USART1;                          //串口 1
    UART1_Handler.Init.BaudRate = 115200;                     //波特率
    UART1_Handler.Init.HwFlowCtl = UART_HWCONTROL_NONE;       //无硬件流控
    UART1_Handler.Init.Mode = UART_MODE_TX_RX;                //收发模式
    UART1_Handler.Init.Parity = UART_PARITY_NONE;             //无奇偶校验
    UART1_Handler.Init.StopBits = UART_STOPBITS_1;            //一个停止位
    UART1_Handler.Init.WordLength = UART_WORDLENGTH_8B;       //字长为 8 位格式
    HAL_UART_Init(&UART1_Handler);                            //初始化串口
}
```

(3) 重定义 HAL_UART_MspInit()函数,初始化串口使用的 GPIO 参数,并使能串口中断。

```
//初始化串口 GPIO
void HAL_UART_MspInit(UART_HandleTypeDef * huart)
{
    if(huart->Instance == USART1)
    {
        GPIO_InitTypeDef GPIO_Initure;

        __HAL_RCC_GPIOA_CLK_ENABLE();                         //使能 GPIO 引脚
        __HAL_RCC_USART1_CLK_ENABLE();                        //使能串口

        GPIO_Initure.Mode = GPIO_MODE_AF_PP;                  //复用推挽模式
        GPIO_Initure.Pin = GPIO_PIN_9|GPIO_PIN_10;            //PA9、PA10
        GPIO_Initure.Pull = GPIO_PULLUP;                      //上拉
        GPIO_Initure.Speed = GPIO_SPEED_FAST;                 //高速
        GPIO_Initure.Alternate = GPIO_AF7_USART1;             //复用为 USART1
        HAL_GPIO_Init(GPIOA, &GPIO_Initure);                  //初始化 PA9、PA10

        __HAL_UART_ENABLE_IT(&UART1_Handler, UART_IT_IDLE);   //使能空闲中断
        __HAL_UART_ENABLE_IT(&UART1_Handler, UART_IT_RXNE);   //使能接收完成中断

        HAL_NVIC_SetPriority(USART1_IRQn, 2, 1);              //设置串口中断优先级
        HAL_NVIC_EnableIRQ(USART1_IRQn);                      //使能串口中断
    }
}
```

(4) 定义串口中断服务函数。函数中分别获取接收寄存器非空、空闲中断的标志位,当检测接收寄存器非空标志时,将串口接收到的字节数据存储到接收缓冲区 buffer 中;当检测到空闲中断标志时,说明整个字符串已经接收完成,这时通过调用 UART_Transmit()发送函数,根据计算的数据长度变量 size,将整个接收的字符串发送出去。

```
uint16_t size;                                                //接收到的数据长度
//串口中断服务函数
void USART1_IRQHandler(void)
{
    //获取接收寄存器非空标志位
    if(__HAL_UART_GET_FLAG(&UART1_Handler, UART_FLAG_RXNE) != RESET)
    {
        buffer[size++] = (uint8_t)UART1_Handler.Instance->DR;     //获取一个字节数据
        __HAL_UART_CLEAR_FLAG(&UART1_Handler, UART_FLAG_RXNE);    //清除标志位
    }

    //获取空闲中断标志位
    if(__HAL_UART_GET_FLAG(&UART1_Handler, UART_FLAG_IDLE) != RESET)
    {
```

```
            UART_Transmit(buffer, size);          //将接收到的数据发送出去
            size = 0;                             //下次接收重新开始计数
        }
    }
```

(5) 定义发送数据函数。函数传入参数为发送数据缓冲区地址和发送数据量的大小。

```
//发送数据函数
void UART_Transmit(uint8_t *pData, uint16_t Size)
{
    HAL_UART_Transmit(&UART1_Handler, pData, Size, HAL_MAX_DELAY);
}
```

(6) 定义接收数据函数。函数传入参数为接收数据缓冲区地址和接收数据量的大小。

```
//接收数据函数
void UART_Receive(uint8_t *pData, uint16_t Size)
{
    HAL_UART_Receive(&UART1_Handler, pData, Size, HAL_MAX_DELAY);
}
```

(7) 在 main.c 文件的主函数中,只需要初始化系统时钟和串口即可。

```
int main(void)
{
    CLOCK_Init();                                 //配置系统时钟为 168MHz
    UART_Init();                                  //串口初始化

    while(1)
    {
    }
}
```

3.2.3 运行结果

将程序下载到开发板中,打开串口。如果此刻发送一个字符串“123456”,将会看到串口助手会返回字符串“123456”,如图 3-8 所示。

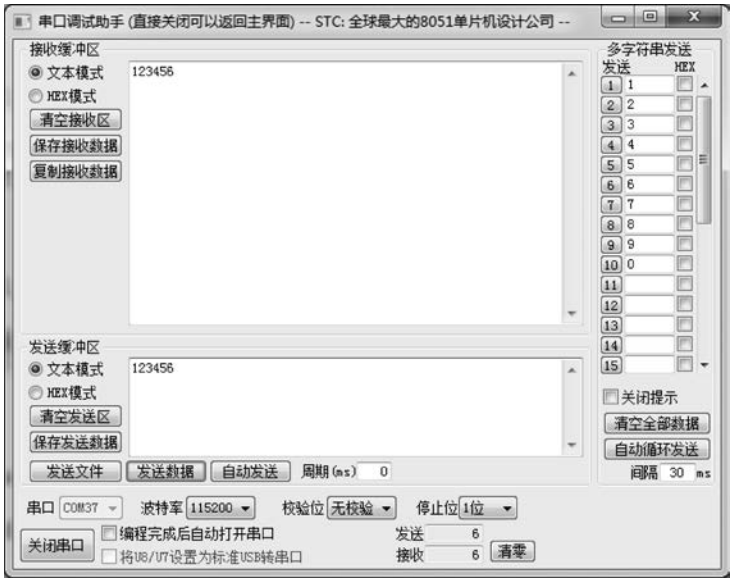

图 3-8 运行结果

练习

(1) 简述串口中断与串口轮询的异同。

(2) 通过本节的学习实现串口发送完成中断。

3.3　串口通信：DMA


视频 9

学习目标

了解 DMA 的工作原理，通过配置 STM32F407 芯片的 DMA，实现串口＋DMA 数据收发。

3.3.1　开发原理

基于 USART 的数据通信中采用中断方式可以在接收信息或发送数据时产生中断，在中断服务程序中完成数据的接收与发送，但是中断方式的 CPU 使用率更高。在简单的系统中，使用中断方式确实是一种好方法。但是在复杂的系统中，处理器需要处理串口通信，多个传感器数据的采集及处理，这涉及多个中断的优先级分配问题。为了保证数据发送与接收的可靠性，需要把 USART 的中断优先级设计得较高，但是系统可能还有其他的需要更高优先级的中断，必须保证其定时的准确，这样就有可能造成串行通信的中断不能及时响应，从而造成数据丢失。为了保证串行通信的数据及时可靠地接收，同时兼顾其他任务不受影响，采用了基于 DMA 和中断方式相结合的 USART 串行通信方式。

DMA 的全称为 Direct Memory Access，即直接存储器访问。DMA 传输将数据从一个地址空间复制到另一个地址空间。DMA 传输方式无须 CPU 直接控制传输，也没有中断处理方式那样的保留现场和恢复现场过程，通过硬件为 RAM 和 I/O 设备开辟一条直接传输数据的通道，使得 CPU 的效率大大提高。DMA 最主要的作用是为 CPU 减小负担。

STM32F4xx 系列的 DMA 功能齐全，工作模式众多，适合不同的编程环境要求。STM32F4xx 系列的 DMA 支持外设到存储器传输、存储器到外设传输和存储器到存储器传输 3 种传输模式。这里的外设一般指外设的数据寄存器，比如 ADC、SPI、I^2C、DCMI 等外设的数据寄存器，存储器一般是指片内 SRAM、外部存储器、片内 Flash 等。

外设到存储器传输就是把外设数据寄存器的内容转移到指定的内存空间。比如进行 ADC 采集时可以利用 DMA 传输将 AD 转换数据转移到我们定义的存储区中，这对于多通道采集、采样频率高、连续输出数据的 AD 采集是非常高效的处理方法。

存储区到外设传输就是将特定存储区内容转移至外设的数据寄存器中，这种多用于外设的发送通信。

存储器到存储器传输就是将一个指定的存储区内容复制到另一个存储区空间。功能类似于 C 语言内存复制函数 memcpy，利用 DMA 传输可以达到更高的传输效率，特别是 DMA 传输是不占用 CPU 的，可以节省很多 CPU 资源。

STM32F4xx 系列的 DMA 可以实现外设与寄存器、存储器之间或者存储器与存储器之间传输 3 种模式，这主要得益于 DMA 控制器是采用 AHB 主总线，可以控制 AHB 总线矩阵来启动 AHB 事务 DMA 控制框图如图 3-9 所示。

1. 外设通道选择

STM32F4xx 系列资源丰富，具有两个 DMA 控制器，同时外设繁多，为实现正常传输，DMA 需要通道选择控制。每个 DMA 控制器具有 8 个数据流，每个数据流对应 8 个外设请求。在实现 DMA 传输之前，DMA 控制器会通过 DMA 数据流 x 配置寄存器 DMA_SxCR(x 为

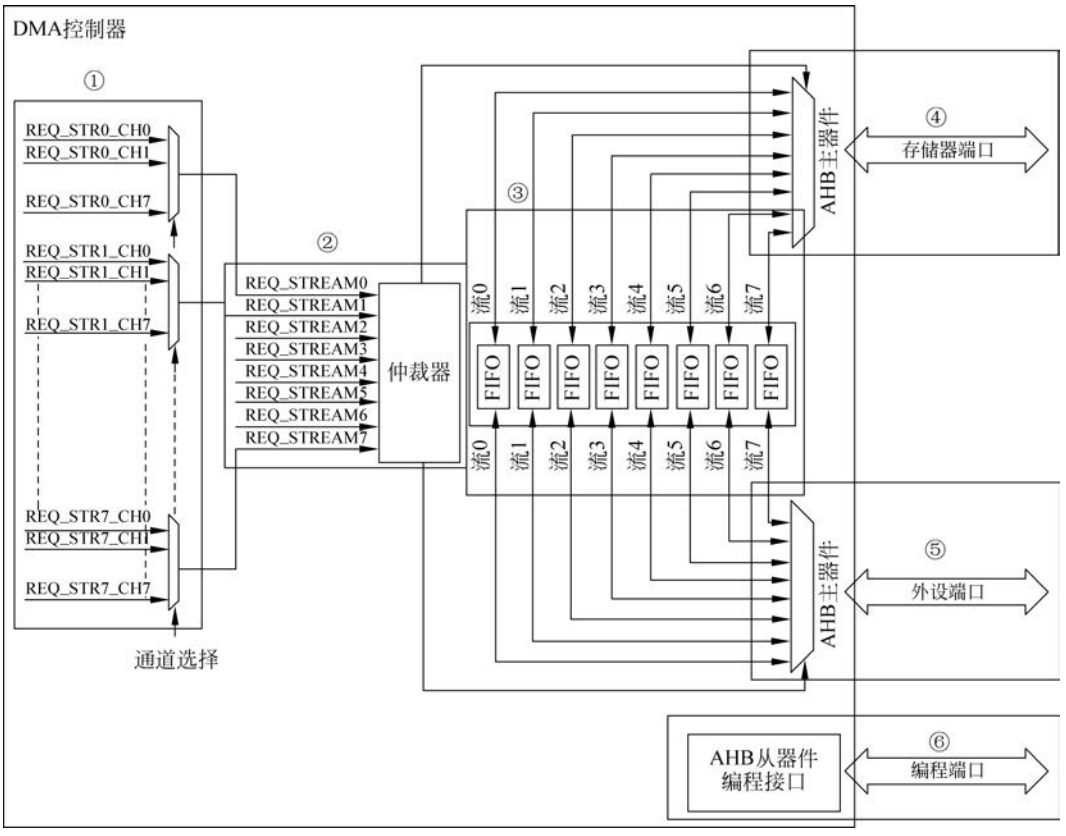

图 3-9　DMA 控制框图

0～7,对应 8 个 DMA 数据流)的 CHSEL[2:0]位选择对应的通道作为该数据流的目标外设。

外设通道选择要解决的主要问题是决定哪一个外设作为该数据流的源地址或者目标地址。

DMA1 请求映射如图 3-10 所示。

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	SPI3_PX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
通道 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
通道 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
通道 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
通道 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
通道 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UARTB_RX(1)	TIM3_CH3
通道 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
通道 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

(1) 这些请求仅在 STM32F42xxx 和 STM32F43xxx 上可用。

图 3-10　DMA1 请求映射

DMA2 请求映射如图 3-11 所示。

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道 1		DCMI	ADC2	ADC2		SPI6_TX(1)	SPI6_RX(1)	DCMI
通道 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
通道 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

(1) 这些请求在 STM32F42xxx 和 STM32F43xxx 上可用。

图 3-11 DMA2 请求映射

2. 仲裁器

一个 DMA 控制器对应 8 个数据流，数据流包含要传输数据的源地址、目标地址、数据等信息。如果需要同时使用同一个 DMA 控制器(DMA1 或 DMA2)处理多个外设请求，那么必然需要同时使用多个数据流，究竟哪一个数据流具有优先传输的权利呢？这就需要仲裁器来管理判断了。

仲裁器管理数据流方法分为两个阶段。第一阶段属于软件阶段，在配置数据流时可以通过寄存器设定它的优先级别，具体配置 DMA_SxCR 寄存器 PL[1:0]位，可以设置为非常高、高、中和低 4 个级别。第二阶段属于硬件阶段，如果两个或以上数据流软件设置优先级一样，则其优先级取决于数据流编号，编号越低越具有优先权，比如数据流 2 优先级高于数据流 3。

3. FIFO

每个数据流都独立拥有 4 级 32 位 FIFO(先进先出存储器缓冲区)。DMA 传输具有 FIFO 模式和直接模式。

直接模式就是每个外设请求都立即启动对存储器传输。在直接模式下，如果 DMA 配置为存储器和外设之间传输，那么 DMA 会将一个数据存放在 FIFO 内。如果外设启动 DMA 传输请求，则可以马上将数据传输过去。

在 FIFO 模式下，FIFO 用于在源数据传输到目标地址之前临时存放这些数据。可以通过 DMA 数据流 x FIFO 控制寄存器 DMA_SxFCR 的 FTH[1:0]位来控制 FIFO 的阈值，分别为 1/4、1/2、3/4 和 1。如果数据存储量达到阈值级别时，FIFO 内容将传输到目标中。

FIFO 对于要求源地址和目标地址数据宽度不同时非常有用，比如源数据是源源不断的字节数据，而目标地址要求输出字宽度的数据，即在实现数据传输时同时把原来 4 个 8 位字节的数据拼凑成一个 32 位数据。此时使用 FIFO 功能先把数据缓存起来，然后根据需要输出数据。

4. 存储器端口、外设端口

DMA 控制器实现双 AHB 主接口，更好利用总线矩阵和并行传输。DMA 控制器通过存储器端口和外设端口与存储器和外设进行数据传输，关系如图 3-12 所示。DMA 控制器的功能是快速转移内存数据，需要一个连接至源数据地址的端口和一个连接至目标地址的端口。

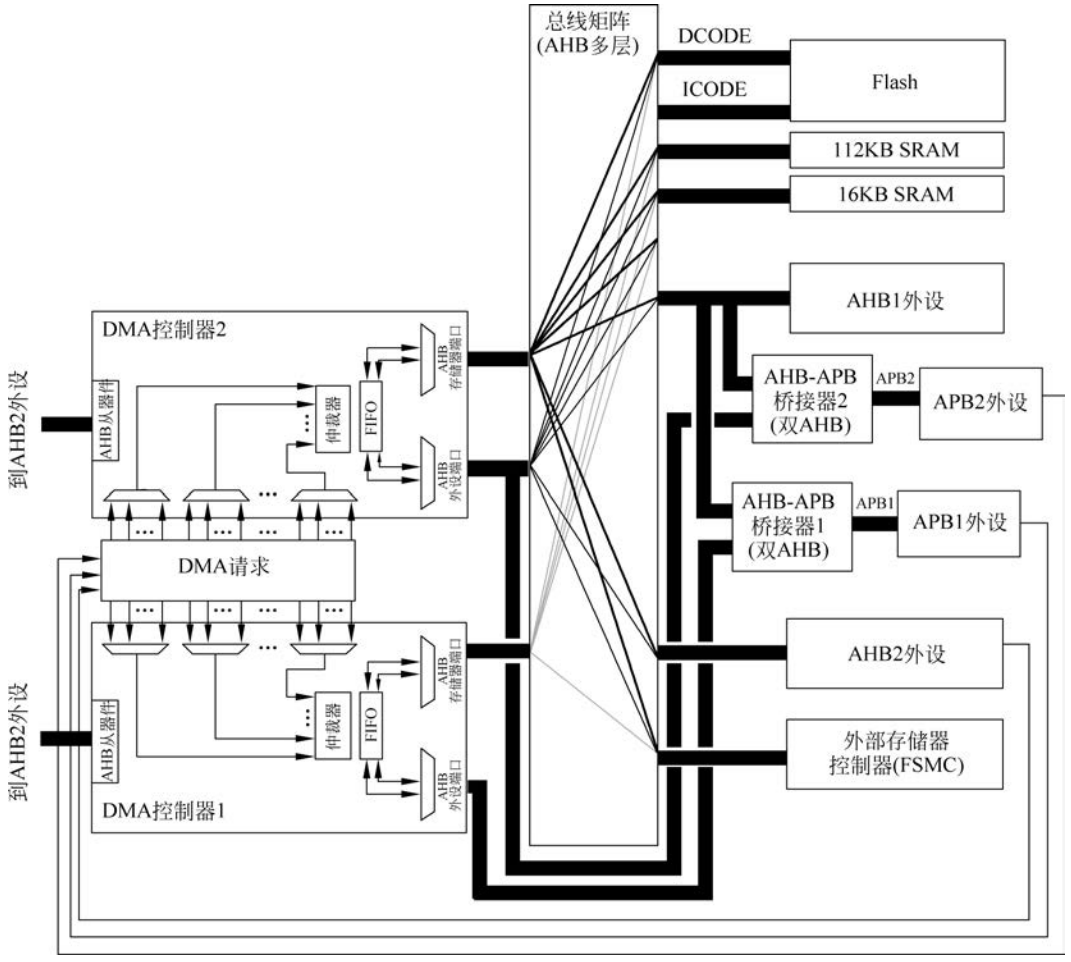

图 3-12　DMA 控制器和外设端口与存储器和外设进行数据传输

DMA2(DMA 控制器 2)的存储器端口和外设端口都是连接到 AHB 总线矩阵，可以使用 AHB 总线矩阵功能。DMA2 存储器和外设端口可以访问相关的内存地址，包括内部 Flash、内部 SRAM、AHB1 外设、AHB2 外设、APB2 外设和外部存储器空间。

DMA1 的存储器端口相比 DMA2 的减少了 AHB2 外设的访问权，同时 DMA1 外设端口是没有连接至总线矩阵的，只连接到 APB1 外设，所以 DMA1 不能实现存储器到存储器的数据传输。

5. 编程端口

AHB 从器件编程端口是连接至 AHB2 外设的。AHB2 外设在使用 DMA 传输时需要相关控制信号。

DMA 寄存器包括 DMA 低中断状态寄存器(DMA_LISR)、DMA 高中断状态寄存器(DMA_HISR)、DMA 低中断标志清除寄存器(DMA_LIFCR)、DMA 高中断标志清除寄存器(DMA_HIFCR)、DMA 流配置寄存器(DMA_SxCR)、DMA 流数据个数寄存器(DMA_

SxNDTR)、DMA 流外设地址寄存器(DMA_SxPAR)、DMA 流内存 0 地址寄存器(DMA_SxM0AR)、DMA 流内存 1 地址寄存器(DMA_SxM1AR)、DMA 流 FIFO 控制寄存器(DMA_SxM1AR)。

3.3.2 开发步骤

(1) 定义串口 DMA 发送和接收的结构体初始化变量。

```
DMA_HandleTypeDef UART1RxDMA_Handler;          //串口接收 DMA 句柄
DMA_HandleTypeDef UART1TxDMA_Handler;          //串口发送 DMA 句柄
```

(2) 定义 UART_DMA_Init()函数,在函数中实现串口 DMA 的初始化。

第一步,使能 DMA2 时钟。

第二步,分别配置 UART1RxDMA_Handler 和 UART1TxDMA_Handler 结构体变量中的参数,然后将两个结构体变量传入到 HAL_DMA_Init()函数中,以便初始化串口 DMA。

第三步,使能 DMA2 数据流 7(串口 DMA 发送)中断。

```
//串口 DMA 初始化
void UART_DMA_Init(void)
{
    __HAL_RCC_DMA2_CLK_ENABLE();                                          //使能 DMA2 时钟

    //接收 DMA 配置
    UART1RxDMA_Handler.Instance = DMA2_Stream5;                           //数据流选择
    UART1RxDMA_Handler.Init.Channel = DMA_CHANNEL_4;                      //通道选择
    UART1RxDMA_Handler.Init.Direction = DMA_PERIPH_TO_MEMORY;             //外设到存储器
    UART1RxDMA_Handler.Init.FIFOMode = DMA_FIFOMODE_DISABLE;              //FIFO 不使能
    UART1RxDMA_Handler.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;      //FIFO 阈值
    UART1RxDMA_Handler.Init.MemBurst = DMA_MBURST_SINGLE;                 //内存突发传输配置
    UART1RxDMA_Handler.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;       //存储器数据长度
    UART1RxDMA_Handler.Init.MemInc = DMA_MINC_ENABLE;     //内存寄存器地址是否自增
    UART1RxDMA_Handler.Init.Mode = DMA_CIRCULAR;          //循环模式
    UART1RxDMA_Handler.Init.PeriphBurst = DMA_PBURST_SINGLE;              //外设突发传输配置
    UART1RxDMA_Handler.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;    //外设数据长度
    UART1RxDMA_Handler.Init.PeriphInc = DMA_PINC_DISABLE;                 //外设地址非自增
    UART1RxDMA_Handler.Init.Priority = DMA_PRIORITY_MEDIUM;               //中等优先级
    HAL_DMA_Init(&UART1RxDMA_Handler);

    //发送 DMA 配置
    UART1TxDMA_Handler.Instance = DMA2_Stream7;                           //数据流选择
    UART1TxDMA_Handler.Init.Channel = DMA_CHANNEL_4;                      //通道选择
    UART1TxDMA_Handler.Init.Direction = DMA_MEMORY_TO_PERIPH;             //外设到存储器
    UART1TxDMA_Handler.Init.FIFOMode = DMA_FIFOMODE_DISABLE;              //FIFO 不使能
    UART1TxDMA_Handler.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;      //FIFO 阈值
    UART1TxDMA_Handler.Init.MemBurst = DMA_MBURST_SINGLE;                 //内存突发传输配置
    UART1TxDMA_Handler.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;       //存储器数据长度
    UART1TxDMA_Handler.Init.MemInc = DMA_MINC_ENABLE;     //内存寄存器地址是否自增
    UART1TxDMA_Handler.Init.Mode = DMA_NORMAL;            //正常模式
    UART1TxDMA_Handler.Init.PeriphBurst = DMA_PBURST_SINGLE;              //外设突发传输配置
    UART1TxDMA_Handler.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;    //外设数据长度
    UART1TxDMA_Handler.Init.PeriphInc = DMA_PINC_DISABLE;                 //外设地址非自增
    UART1TxDMA_Handler.Init.Priority = DMA_PRIORITY_MEDIUM;               //中等优先级
    HAL_DMA_Init(&UART1TxDMA_Handler);

    HAL_NVIC_SetPriority(DMA2_Stream7_IRQn, 0, 0);
```

```
    HAL_NVIC_EnableIRQ(DMA2_Stream7_IRQn);
}
```

(3) 定义 DMA2 数据流 7(串口 DMA 发送)中断服务函数,在函数中调用 HAL_DMA_IRQHandler()处理 DMA 中断请求。

```
//缓冲区 -> Tx 中断服务函数
void DMA2_Stream7_IRQHandler(void)
{
    HAL_DMA_IRQHandler(&UART1TxDMA_Handler);
}
```

(4) 定义串口 DMA 接收数据函数。在函数中调用 HAL_UART_Receive_DMA()实现使用 DMA 从串口接收数据。

```
//串口 DMA 接收数据
void UART_DMA_Receive(uint8_t * pData, uint16_t Size)
{
    HAL_UART_Receive_DMA(&UART1_Handler, pData, Size);
}
```

(5) 定义串口 DMA 发送数据函数。在函数中首先调用 HAL_DMA_Start_IT()开启串口 DMA 发送完成中断,并设置发送数据缓冲区地址、外设寄存器地址以及发送数据的个数,然后调用 SET_BIT 配置串口 DMA 发送标志位(DMAT)。

```
extern uint8_t Rx_buffer[RXBUFFERSIZE]; //发送数据缓冲区

//串口 DMA 发送数据
void UART_DMA_Transmit(UART_HandleTypeDef * huart)
{
    HAL_DMA_Start_IT(UART1_Handler.hdmatx, (uint32_t)Rx_buffer, (uint32_t)&UART1_Handler.
Instance->DR, RXBUFFERSIZE);
SET_BIT(UART1_Handler.Instance->CR3, USART_CR3_DMAT);
}
```

(6) 修改串口初始化函数,将串口 DMA 发送和接收初始化句柄传入串口初始化结构体变量中,具体实现如下:

```
//初始化串口函数
void UART_Init(void)
{
    UART1_Handler.Instance = USART1;                        //串口 1
    UART1_Handler.Init.BaudRate = 115200;                   //波特率
    UART1_Handler.Init.HwFlowCtl = UART_HWCONTROL_NONE;     //无硬件流控制
    UART1_Handler.Init.Mode = UART_MODE_TX_RX;              //收发模式
    UART1_Handler.Init.Parity = UART_PARITY_NONE;           //无奇偶校验
    UART1_Handler.Init.StopBits = UART_STOPBITS_1;          //一个停止位
    UART1_Handler.Init.WordLength = UART_WORDLENGTH_8B;     //字长为 8 位格式
    UART1_Handler.hdmarx = &UART1RxDMA_Handler;             //传入接收 DMA 句柄
    UART1_Handler.hdmatx = &UART1TxDMA_Handler;             //传入发送 DMA 句柄
    HAL_UART_Init(&UART1_Handler);                          //初始化串口
}
```

(7) 在 main.c 文件中调用函数实现串口 DMA 数据收发。

第一步,声明串口初始化句柄方便下面函数调用,并定义接收数据缓冲区。

第二步,初始化系统时钟、DMA 及串口。

第三步,在循环中调用 UART_DMA_Transmit 和 HAL_Delay()函数,实现每隔 1s 发送

一组收到的数据。

```
extern UART_HandleTypeDef UART1_Handler;                //UART 句柄
uint8_t Rx_buffer[RXBUFFERSIZE];                        //串口接收数据缓冲区

int main(void)
{
    CLOCLK_Init();                                      //配置系统时钟为 168MHz
    UART_DMA_Init();                                    //串口 DMA 初始化
    UART_Init();                                        //串口初始化

    UART_DMA_Receive(Rx_buffer, RXBUFFERSIZE);          //使用 DMA 接收数据

    while(1)
    {
        HAL_Delay(1000);                                //延时 1s
        UART_DMA_Transmit(&UART1_Handler);              //发送数据
    }
}
```

3.3.3　运行结果

将程序下载到开发板中，打开串口。将发送的数据添加到发送缓冲区，单击自动发送按钮，可以看到接收缓冲区每隔 1s 会接收到返回的数据，如图 3-13 所示。

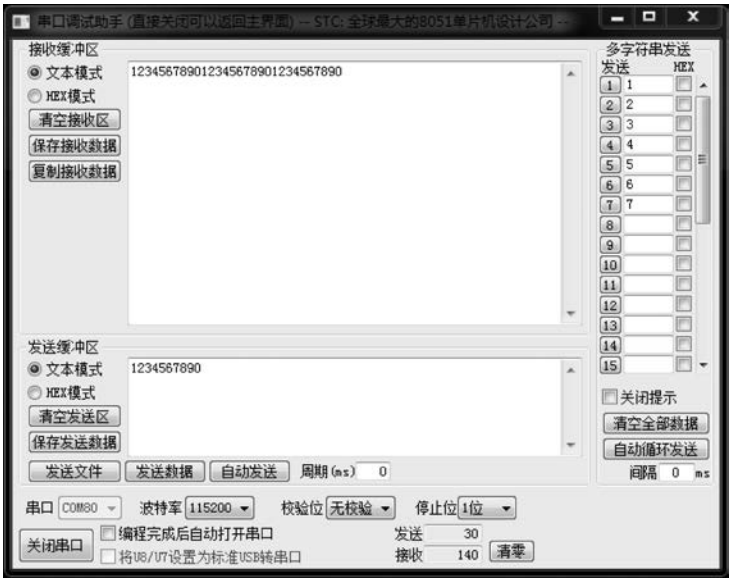

图 3-13　运行结果

练习

（1）什么是 DMA？

（2）以 DMA 的方式配置其他串口，实现数据接收与发送。